

# THESE

*présentée*

A LA FACULTÉ DES SCIENCES DE L'UNIVERSITÉ PAUL-SABATIER DE TOULOUSE

*pour obtenir*

LE GRADE DE DOCTEUR-ES-SCIENCES MATHÉMATIQUES

*par*

Bernard LORHO

DE LA DÉFINITION A LA TRADUCTION  
DES LANGAGES DE PROGRAMMATION :  
MÉTHODE DES ATTRIBUTS SÉMANTIQUES

Soutenue le 29 novembre 1974, devant la Commission d'Examen :

MM. M. LAUDET

Président

C. PAIR  
R. BEAUFILS  
M. GRIFFITHS  
F.L. DE REMER  
J.J. DUBY

} Examinateurs



# THESE

*présentée*

A LA FACULTÉ DES SCIENCES DE L'UNIVERSITÉ PAUL-SABATIER DE TOULOUSE

*pour obtenir*

LE GRADE DE DOCTEUR-ES-SCIENCES MATHÉMATIQUES

*par*

Bernard LORHO

DE LA DÉFINITION A LA TRADUCTION  
DES LANGAGES DE PROGRAMMATION :  
MÉTHODE DES ATTRIBUTS SÉMANTIQUES

Soutenue le 29 novembre 1974, devant la Commission d'Examen :

|               |              |
|---------------|--------------|
| MM. M. LAUDET | Président    |
| C. PAIR       | } Examineurs |
| R. BEAUFILS   |              |
| M. GRIFFITHS  |              |
| F.L. DE REMER |              |
| J.J. DUBY     |              |

THESE

UNIV.

LEIDEN

BIBL.

UNIV.

LEIDEN



UNIV.

LEIDEN

BIBL.

UNIV.

LEIDEN

BIBL.

UNIV.

LEIDEN

BIBL.

Monsieur le Professeur LAUDET a bien voulu s'intéresser à mon travail depuis de nombreuses années, et ses encouragements chaleureux n'ont jamais cessé. Je le prie d'accepter mes remerciements très sincères.

La rédaction finale de cette thèse doit beaucoup à Monsieur le Professeur PAIR. Qu'il soit assuré de ma gratitude, et je ne peux que souhaiter à tous les chercheurs d'avoir un correcteur de thèse aussi vigilant et encourageant.

Je remercie Monsieur le Professeur BEAUFILS d'avoir bien voulu faire partie de mon jury en espérant que les prolongements en matériel de la conclusion de cette étude trouveront une application dans ses travaux.

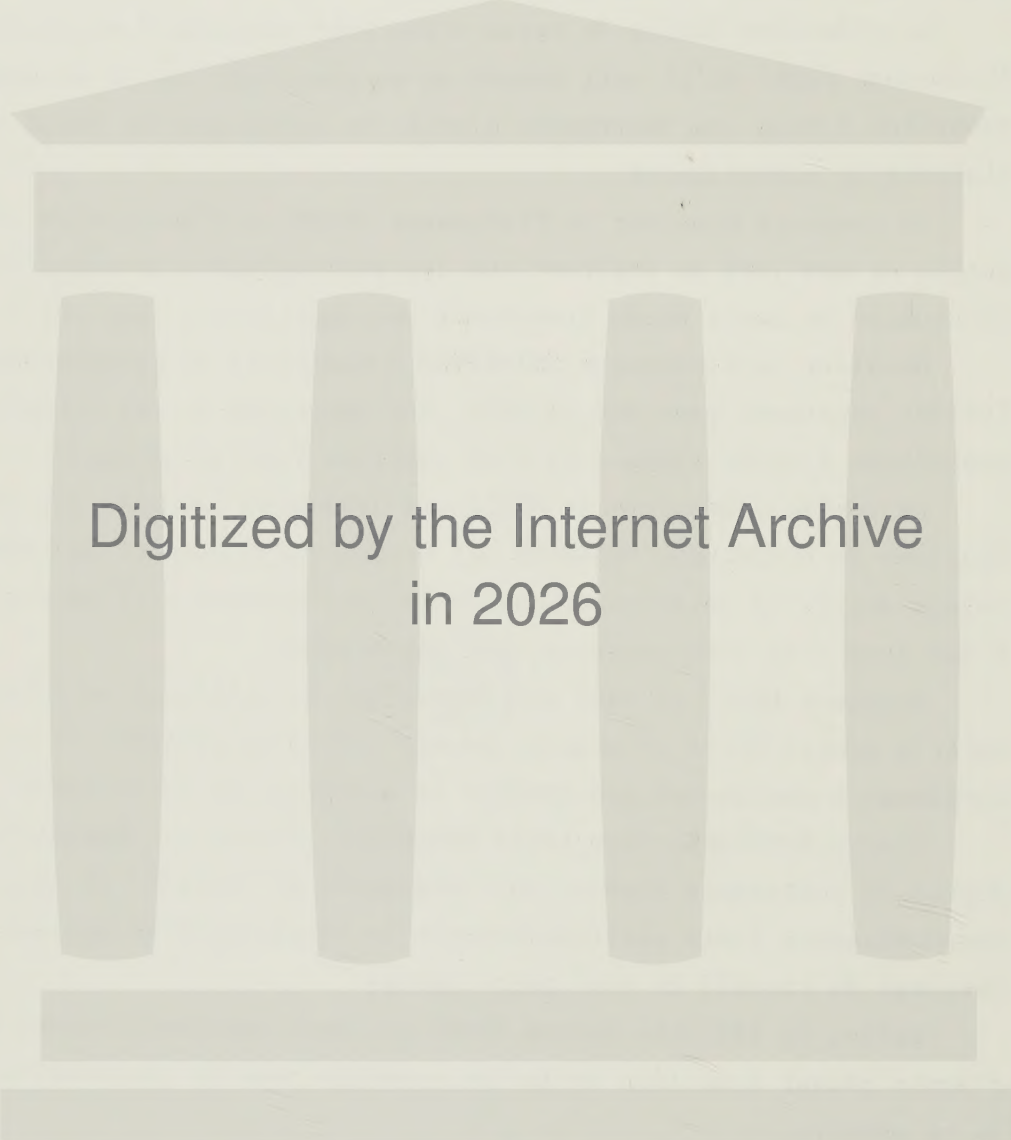
Monsieur le Professeur GRIFFITHS a manifesté un enthousiasme et un intérêt constants pour mon travail. Ses remarques et ses conseils ont contribué à cette thèse : je l'en remercie très amicalement.

La partie syntaxique de DELTA est largement inspirée des travaux de Monsieur le Professeur DE REMER qui a bien voulu avancer un très long voyage depuis la Californie pour me faire l'honneur et l'amitié de participer à mon jury : je l'en remercie très sincèrement.

Monsieur DUBY, en tant que conseiller scientifique de l'IRIA, a eu foi dans le projet DELTA et a ainsi permis son développement. Je le remercie de continuer à manifester son intérêt en acceptant de faire partie de mon jury.

Pierre BOULLIER, Jean Louis BOUCHENEZ, France et Laurent BLAIZOT sont, depuis de nombreuses années, mes compagnons de travail. Je leur dois une reconnaissance toute particulière car la réalisation du système DELTA est le résultat du travail de tout notre groupe.

Enfin, je félicite Madame PEREZ et, tout particulièrement Madame SUARD, d'avoir réussi à me lire et je les remercie pour la réalisation matérielle de ce mémoire.



Digitized by the Internet Archive  
in 2026

[https://archive.org/details/IA41546613\\_0012](https://archive.org/details/IA41546613_0012)

*à Danièle*

*à Marie-Christine et François*



## TABLE DES MATIERES

|                                                                               | <u>pages</u> |
|-------------------------------------------------------------------------------|--------------|
| INTRODUCTION                                                                  |              |
| <u>Première Partie : METHODES DE DEFINITION DES LANGAGES DE PROGRAMMATION</u> |              |
| 1. DEFINITION DES LANGAGES DE PROGRAMMATION                                   | 1            |
| 1. Problème général                                                           |              |
| 2. Définition de E                                                            |              |
| 3. Définition de S                                                            |              |
| 1. <i>Sémantique dénotationnelle</i>                                          |              |
| 2. <i>Sémantique opérationnelle</i>                                           |              |
| 3. <i>Sémantique axiomatique ou déductive</i>                                 |              |
| 4. <i>Lambda - expressions</i>                                                |              |
| 5. <i>Sémantique traductive</i>                                               |              |
| 4. Définition de f                                                            |              |
| 1. <i>Méthodes interprétatives</i>                                            |              |
| 2. <i>Méthodes traductives</i>                                                |              |
| 2. DEFINITION DU TYPE ALGOL 60                                                | 9            |
| 1. Présentation de TURINGOL                                                   |              |
| 2. Description formelle "à la ALGOL 60"                                       |              |
| 3. Critiques de la méthode                                                    |              |
| 1. <i>Critiques des théoriciens</i>                                           |              |
| 2. <i>Critiques des usagers</i>                                               |              |
| 3. <i>Critiques des implanteurs</i>                                           |              |
| 4. Intérêt de la méthode                                                      |              |
| 5. Définition de TL/1                                                         |              |
| 1. <i>Présentation</i>                                                        |              |
| 2. <i>Définition d'un interprète de TL/1</i>                                  |              |
| 3. LANGAGE DE DEFINITION DE VIENNE (VDL)                                      | 15           |
| 1. Introduction                                                               |              |
| 2. Syntaxe concrète, syntaxe abstraite                                        |              |
| 1. <i>Introduction</i>                                                        |              |
| 2. <i>Syntaxe abstraite de TURINGOL</i>                                       |              |
| 3. Etat de l'interprète                                                       |              |
| 4. Description de la librairie                                                |              |
| 5. Critiques de la méthode                                                    |              |

1. *Critiques des théoriciens*
2. *Critiques des usagers*
3. *Critiques des implanteurs*

#### 4. DEFINITION AXIOMATIQUE OU DEDUCTIVE

22

1. Introduction
2. Présentation de la méthode
3. Définition de TURINGOL - bis
4. Preuve de correction d'un programme
5. Critiques de la méthode

#### 5. DEFINITION GRAMMATICALE DE VAN WIJNGAARDEN

29

1. Introduction
2. Définitions
  1. *Grammaire de premier niveau*
  2. *Grammaire de deuxième niveau*
  3. *Terminologie*
  4. *Commentaires*
  5. *Exemple 1*
  6. *Exemple 2*
3. Définition des règles grammaticales de TURINGOL
4. Critiques de la méthode
5. W-grammaires et sémantique

#### 6. METHODES ALGORITHMIQUES DE DEFINITION TRANDUCTIVE

39

1. Introduction
2. Méthode de Wirth et Weber
  1. *Principe*
  2. *Définition de TURINGOL*
  3. *Critiques de la méthode*
3. Méthode des grammaires affixes
  1. *Introduction*
  2. *Principe*
  3. *Exemples*
  4. *Critiques de la méthode*

## 7. METHODE DECLARATIVE DE DEFINITION TRADUCTIVE

52

1. Introduction
2. Présentation informelle de la méthode
3. Intérêt de la méthode
4. Critiques de la méthode

## 8. COMPARAISON ENTRE ATTRIBUTS SEMANTIQUES, DOUBLES GRAMMAIRES ET GRAMMAIRES AFFIXES

60

1. Introduction
2. Comparaison informelle
  1. *Type de définition*
  2. *Flot d'informations contextuelles*
  3. *Synthèse et héritage d'informations*
  4. *Ordre du flot*
3. Comparaison des méthodes deux à deux
  1. *Attributs sémantiques et grammaires affixes*
  2. *W-grammaires et attributs sémantiques*
  3. *W-grammaires et grammaires affixes*
4. Schéma général de correspondance

## Deuxième Partie : METHODE DES ATTRIBUTS SEMANTIQUES, SYSTEME DELTA

## 9. NOTATIONS SYNTAXIQUES ET SEMANTIQUES, NOTION DE COHERENCE D'UN SYSTEME D'EQUATIONS

69

1. Introduction
2. Notations et définitions syntaxiques
3. Notations et définitions sémantiques
  1. *Définitions préliminaires*
  2. *Notations et définitions*
4. Exemples
5. Notion de décoration d'un arbre syntaxique
  1. *Présentation*
  2. *Opération de décoration*
  3. *Exemple*
6. Relations de dépendance sur un système d'équations
7. Cohérence d'un système d'équations
8. Langages à assignation unique (LAU)
9. Système d'équations, langages à assignation unique

1. Introduction
2. Notion de cohérence
3. Définition unique
4. Non circularité
  1. *Définitions*
  2. *Exemples*
5. Premier algorithme (A1)
  1. *Algorithme*
  2. *Exemple*
6. Deuxième algorithme
  1. *Première version (A21)*
  2. *Deuxième version (A22)*
7. Troisième algorithme (A3)
8. Quatrième algorithme (A4)
  1. *Elimination des relations inutiles*
  2. *Première itération*
  3. *Algorithme A4*
  4. *Exemple*
  5. *Applicabilité de l'algorithme A4*

## 11. SYSTEME DELTA

1. Organisation générale d'un métacompilateur
2. Traitement à la construction
  1. *Constructeurs syntaxiques de DELTA*
  2. *Constructeur sémantique de DELTA*
3. Traitement à l'analyse syntaxique
  1. *Décoration des ramifications*
  2. *Equations sémantiques*
4. Généralités sur l'évaluation sémantique
  1. *Introduction*
  2. *Evaluation sémantique par processus coopérants*
  3. *Evaluation sémantique par recherche d'un o-dre de résolution*
  4. *Evaluation sémantique en alternance avec l'analyse syntaxique gauche - droite*
5. Réduction du système d'équations
  1. *Elimination des identités*
  2. *Elimination des impasses*

6. Traitement à l'évaluation sémantique dans DELTA
  1. Définitions
  2. Tri descendant
  3. Tri ascendant
  4. Comparaison entre tri descendant et ascendant
7. Espace mémoire nécessaire à l'évaluation sémantique
  1. Définitions
  2. Exemple
  3. Solutions optimales
  4. Allocation des tris descendant et ascendant
8. Résultats expérimentaux
9. Mise en oeuvre de DELTA
  1. Présentation fonctionnelle
  2. Restriction actuelle
  3. Caractéristiques du système

## 12. COMPARAISON DES SYSTEMES A ATTRIBUTS, PERSPECTIVES D'AVENIR

143

1. Introduction
2. Systèmes à attributs
3. Comparaison de DELTA à FOLDS et SOFT
  1. Construction syntaxique
  2. Construction sémantique
  3. Evaluation sémantique
4. Langages dirigés par les données
5. Exécution séquentielle de programmes dirigés par les données
6. Exécution parallèle de programmes dirigés par les données
7. Compilation parallèle

## CONCLUSION

## BIBLIOGRAPHIE

## ANNEXES

- Annexe 1 : TRADUCTION DE PROGRAMMES CONCRETS EN PROGRAMMES ABSTRAITS
- Annexe 2 : INTERPRETEUR ABSTRAIT DE TURINGOL COMPLET



## INTRODUCTION

La définition d'un langage de programmation concerne trois catégories de personnes : les théoriciens, les utilisateurs et les implanteurs. Une situation idéale serait de trouver une méthode de définition satisfaisante pour tout le monde. Malheureusement, les exigences des uns et des autres sont pratiquement inconciliables.

Les théoriciens attendent d'une définition, un outil de déduction des propriétés des programmes. Pour cela, il souhaitent transformer un programme en une représentation abstraite de concepts mathématiques qu'ils peuvent manipuler pour en extraire les propriétés recherchées.

Un autre aspect important, pour les théoriciens, est qu'une définition exprime la puissance d'un langage de manière moins intuitive que la ressentent les utilisateurs, pour qui cette puissance s'évalue par les possibilités d'un langage par rapport à d'autres langages.

Les utilisateurs recherchent, dans la définition d'un langage, les mêmes effets que les théoriciens, c'est-à-dire souhaitent avoir la preuve que leurs programmes réalisent bien les fonctions pour lesquelles ils sont écrits. Ils souhaitent également être aidés et guidés dans la rédaction des programmes pour réaliser les fonctions nécessaires à leurs besoins.

Ces deux points de vues sont donc assez proches : pour le théoricien et l'utilisateur, un programme réalise une fonction que le premier préfère considérer comme mathématique et le second regarde d'un oeil plus pratique. Mais la démarche est bien la même. Si on veut un exemple, considérons la notion de programmation structurée. Elle a été introduite par les théoriciens car elle simplifie et systématise les règles de déduction des propriétés des programmes. Il semble surprenant que, pour la majorité des utilisateurs, programmation structurée soit synonyme de contraintes sévères, dont on ne leur a pas expliqué les motivations profondes. Une méthodologie de la programmation, si elle a des justifications pratiques comme la lisibilité des programmes, la clarté de rédaction, une plus grande fiabilité, a des résonances plus profondes qui sont mal ressenties : permettre et faciliter des preuves de corrections. Et ce n'est pas le bannissement sans pitié des instructions de branchement qui a éclairé le malentendu alors que les utilisateurs ressentent la notion de branchement comme fondamentale.

Le vrai débat est, bien sûr, au niveau de la sémantique des langages.

Nous avons voulu donner un exemple des perturbations qu'introduit l'instruction de branchement dans la définition d'un langage (§ 3.4 et annexe 2, ou chapitre 4). Nous demandons aux utilisateurs qui estiment que la suppression des instructions de branchement n'est qu'une question d'esthétique de comparer ces deux paragraphes et de méditer. Et nous demandons aux théoriciens de ne pas proscrire tous les concepts qui ne se plient pas à leurs modèles, sans en avoir mesurer les conséquences. La troisième catégorie de personnes intéressées à une définition de langages nous est chère : ce sont les planteurs. Pour eux, la définition d'un langage est une loi impérative et absolue qu'il doivent mettre en application. La solution la plus élégante consiste, bien sûr, à déduire directement l'implantation de la définition, un passage automatique étant la solution idéale. Si on s'intéresse aux méthodes de définition existantes, on réalise la difficulté de ce passage. Entre les objets mathématiques et les règles de déduction des définitions théoriques, d'une part, les objets informatiques et les possibilités des calculateurs, d'autre part, il n'y a rien de commun. Les planteurs ont résolu cette difficulté en envisageant deux types d'implantation :

- . interprétative, où les objets mathématiques sont représentés et manipulés par des programmes qui sont la transcription des règles de déduction,
- . traductive, où le passage est décomposé en deux parties : la première où le programme est traduit en tenant compte partiellement des règles de déduction ; la deuxième où l'exécution du programme traduit applique les règles de déduction qui n'ont pas été considérées au cours de la traduction.

Ces deux approches sont bien connues, et, notre intérêt marqué pour la seconde se justifie par les lacunes qui y subsistent et qui sont importantes (voir chapitre 6) et par des raisons économiques évidentes : l'utilisation de traducteurs est permanente dans la vie informatique. Toute amélioration des méthodes de traduction ne peut donc qu'être bénéfique.

Les objectifs de cette thèse peuvent maintenant être définis : nous sommes, tout d'abord, intéressés à toutes les méthodes de définition des langages de programmation, mais nous demandons à ces méthodes de permettre un passage automatique au traducteur correspondant. Ensuite, ayant choisi la méthode qui nous semble la mieux adaptée à notre problème, nous nous intéressons à ses avantages sur les méthodes classiques de traduction, et aussi à ses inconvénients. Nous définissons ensuite les moyens informatiques qui permettent de réduire ces inconvénients.

Tous ces moyens ont été implantés dans un système nommé DELTA (pour système de DEfinition de Langages et de Traducteurs par Attributs) dont nous analysons les performances.

Notre souhait est de convaincre les planteurs que les méthodes utilisées quotidiennement peuvent être largement améliorées par des preuves de cohérence du traducteur obtenu, par une minimisation de la traduction et par des perspectives tout à fait nouvelles sur la traduction effectuée en parallèle sur une structure de multi-processeurs.

Cette thèse comprend deux parties. La première partie est consacrée à certaines méthodes existantes de définition de langage de programmation (chapitre 1 à 8).

Le premier chapitre est un essai de classification de ces méthodes et est suivi de la présentation rapide mais détaillée de certaines d'entre elles. Nous avons tenu à cette présentation car l'étude séparée de méthodes différentes montrent quelquefois mal ce qui les différencie ou ce qui les rapproche. Pour aider le lecteur et pour lui offrir une base de comparaison, nous avons tenu à définir, au moins partiellement, le même langage. Nous pensons ainsi limiter l'écueil que constitue le choix de langages particulièrement adaptés à une méthode mais qui le seraient moins à d'autres, et aider ainsi à cerner les limites des unes et des autres.

Nous avons, en fait, ainsi reconstitué une partie de l'étude bibliographique que nous avons faite au début de ce projet, en estimant dommage qu'elle ne serve pas à d'autres que nous.

Il n'était, bien sûr, pas question dans cette première partie, de présenter toutes les méthodes existantes ; nous avons fait un choix, certainement critiquable.

Nous ne prétendons pas, pour les méthodes décrites, en donner tous les détails ; nous avons essayé d'en dégager les idées essentielles, en espérant y avoir réussi sans les avoir trahies.

La deuxième partie de cette thèse est consacrée à la méthode de définition de langages par attributs sémantiques. Les propriétés de cette méthode sont étudiées ; en particulier, la notion de cohérence est définie et des algorithmes de preuve de cohérence sont établis. Les principes de conception du système DELTA sont définis et découlent des propriétés de la méthode choisie. Nous montrons que ces principes dépassent largement le cadre d'un système de métacompilation et sont, également, à la base de langages dirigés par les données et se retrouvent, aussi, dans la résolution informatique des systèmes d'équations. Tous ces problèmes sont liés et trouvent une solution naturelle et efficace sur une structure de calculateurs multiprocesseurs.

Nous donnons des algorithmes pour cette structure de machines et pour la structure traditionnelle monoprocesseur.

## Première Partie

METHODES DE DEFINITION DES LANGAGES DE PROGRAMMATION



## 1. DEFINITION DES LANGAGES DE PROGRAMMATION

### 1.1. Problème général.

Ce chapitre est consacré à des notions générales sur la définition d'un langage de programmation.

De façon tout à fait intuitive, qu'est ce qu'un langage de programmation ? C'est un outil de description de données, et de manipulation de ces données en vue d'un certain traitement.

Ecrire un programme dans un langage met donc en oeuvre trois notions liées :

- l'écriture dans le langage,
- l'ensemble des significations possibles,
- la correspondance entre l'écriture et la signification.

Plus formellement, on peut donc définir un langage  $L$  comme une fonction  $f$  (la correspondance) entre un ensemble de départ  $E$  (l'ensemble des programmes écrits) et un ensemble d'arrivée  $S$  (l'ensemble des significations ou valeurs sémantiques).

Définir un langage revient donc à définir l'ensemble  $E$ , l'ensemble  $S$  et la fonction  $f$  de  $E$  dans  $S$ . Selon les modèles de définition choisis, l'accent peut être plus particulièrement mis sur l'un de ces trois aspects, mais il est certain qu'alors, les autres aspects sont implicitement supposés connus. Ce chapitre va passer en revue certains modèles de définition des trois aspects fondamentaux pour un langage de programmation et aiguiller vers les chapitres suivants, qui sont consacrés à certaines méthodes particulières, ou vers la bibliographie.

### 1.2. Définition de $E$

L'ensemble de départ,  $E$ , permet de définir la syntaxe du langage  $L$ , c'est-à-dire les règles qui permettent d'écrire correctement un programme dans le langage.

La définition de  $E$  se fait, depuis Chomsky [16], par la définition des structures syntaxiques du langage  $L$  et, on utilise, le plus souvent, une grammaire hors-contexte pour mettre en évidence ces structures. Cette approche se justifie, tout à fait, pour les langages non contextuels. En fait, les langages usuels sont contextuels et un modèle hors contexte est insuffisant pour décrire certains aspects syntaxiques. Ainsi, la plupart des langages comportent des déclarations de symboles associés à certains types de données,

informations qui doivent être accessibles depuis tout ou partie du programme. De par sa nature, une grammaire hors contexte ne permet pas d'exprimer l'accès à ces informations. D'autre part, les structures syntaxiques décrites par une telle grammaire, définissent, en général, un sur-ensemble de E auquel il convient d'apporter des restrictions pour le conformer au langage que l'on veut définir. Nous reviendrons sur ce point à de nombreuses reprises.

Ces difficultés s'accompagnent de deux aspects quelquefois gênants de l'utilisation de grammaires hors contexte : tout d'abord, l'apparition d'informations intermédiaires inutiles qui viennent de raisons purement grammaticales et non de définition. Citons, par exemple, dans la définition d'une expression arithmétique, l'apparition de notions intermédiaires comme terme, facteur, primaire, opérande arithmétiques qui n'apportent rien du point de vue de la définition. Ensuite, un autre aspect gênant est lié à la définition des programmes à un niveau très concret qui masque souvent les entités abstraites que l'on veut mettre en évidence. Citons, par exemple les différentes formes possibles d'écriture d'une même entité qui font que les concepts importants sont noyés dans un "sucre syntaxique", en suivant Strachey. Pour lever ces difficultés, on peut être conduit à prendre comme ensemble E, non l'ensemble des programmes mais un ensemble de "structures profondes" au sens de Chomsky.

Ces nombreuses critiques justifient la recherche de modèles de définition plus puissants. Les premières difficultés soulevées - informations contextuelles à transporter et adéquation au langage défini - sont résolues au chapitre 5 par une méthode de description grammaticale à deux niveaux. Le deuxième type de difficultés - suppression d'informations inutiles et abstraction des entités rencontrées dans un programme - est abordé dans la première partie du chapitre 3, par l'utilisation d'une syntaxe abstraite. Mais toutes ces critiques ne justifient pas l'abandon pur et simple des modèles hors-contexte pour la définition de E, car les facilités analytiques de ces modèles sont importantes. Nous y reviendrons au chapitre 2.

### 1.3. Définition de S

L'ensemble S représente les propriétés sémantiques, c'est-à-dire la signification des instructions du langage L.

1.3.1. Une première approche, la *sémantique dénotationnelle* ([19], [20], [33], [62]) considère que la signification d'un programme est une fonction d'un ensemble de données dans un ensemble de résultats. Le point essentiel de cette méthode est de définir la fonction mathématique que dénote un programme. L'expression de la sémantique d'un programme est statique, par opposition à

certaines méthodes qui ne peuvent en donner qu'une expression dynamique par le déroulement de l'algorithme que représente le programme.

La dénotation mathématique de la fonction réalisée par un programme offre l'énorme avantage de ne mettre en oeuvre aucun procédé algorithmique mais présente le sérieux inconvénient de masquer certaines réalités informatiques : On peut imaginer deux programmes qui dénotent la même fonction mathématique, mais qui, du point de vue des algorithmes qu'ils représentent, sont de complexité très différente en temps de calcul et / ou en occupation mémoire. La notion de calcul est cachée, la fonction ignorant l'algorithme qui réalise le calcul.

Nous n'étudierons pas de telles méthodes, ici, car elles nous éloignent notablement des préoccupations informatiques qui sont les nôtres. De plus, une définition sémantique de notre langage témoin a été réalisée par cette méthode [37], ainsi qu'une comparaison avec la méthode des attributs sémantiques qui est au centre de cette thèse.

1.3.2. Une deuxième approche, *la sémantique opérationnelle* est caractéristique des méthodes impératives et algorithmiques. Ces méthodes considèrent que la sémantique d'une instruction est définie par l'effet de cette instruction sur un ensemble d'informations qui représentent la situation du calcul exprimé par le programme, avant l'exécution de cette instruction. Ces informations constituent l'état du calcul. La sémantique d'une instruction est alors définie par une fonction de transition entre états.

L'ensemble des états successivement atteints par des transitions depuis un état initial, constitue une trace de l'exécution d'un programme.

La méthode de Vienne, étudiée au chapitre 3, est caractéristique de l'approche opérationnelle, la fonction de transition entre états étant décrite algorithmiquement par le fonctionnement d'un interprète abstrait.

1.3.3. Une troisième approche, *la sémantique axiomatique ou déductive*, est étudiée au chapitre 4.

Dans cette méthode, plutôt que de mémoriser les données dans un état, sous forme d'une structure de données informatiques, les propriétés des données sont mises en évidence sous forme de prédicats logiques. Ces propriétés sont établies comme des théorèmes à partir d'axiomes, chaque instruction créant de nouveaux théorèmes dans un système logique. La sémantique d'un programme est donc établie, de proche en proche, à partir d'axiomes dérivés des données, par une succession de formules logiques qui définissent le théorème final sur la fonction réalisée par le programme.

Cette méthode est donc particulièrement précieuse pour vérifier qu'un programme est correct, relativement à certaines hypothèses incorporées sous forme d'assertions dans la preuve de correction.

On doit remarquer qu'elle ne diffère pas fondamentalement de la deuxième approche. Dans un cas, la définition est donnée par un objet de type informatique, le résultat associé à un état de départ étant obtenu par un calcul réalisé par une machine abstraite ; dans l'autre cas, la définition est donnée par un objet mathématique, le résultat associé à une assertion de départ étant une autre assertion obtenue par une démonstration à partir d'un système logique. Dans les deux cas, on a un interprète qui établit le résultat.

Mais une différence importante doit être notée, malgré tout, entre ces deux approches : l'approche opérationnelle déroule dynamiquement un calcul alors que l'approche déductive établit statiquement les propriétés d'un programme, sans exécution effective de ce programme.

1.3.4. L'ensemble  $S$  peut être défini de bien d'autres manières. En restant dans les systèmes formels, on peut, par exemple, utiliser les lambda expressions [6]. Nous n'étudierons pas cette approche ici. Deux modèles pour des langages inspirés d'ALGOL60 peuvent être trouvés en [15] et [63].

1.3.5. L'ensemble  $S$  peut, aussi, être défini par un autre langage. Cette approche *traductive* est typiquement informatique car le langage d'arrivée peut être celui d'un calculateur ; elle peut être rapprochée de l'approche opérationnelle qui remplace un calculateur particulier par une machine abstraite.

Les méthodes traductives sont au centre de cette thèse et sont abordées aux chapitres 6 et 7 et la méthode que nous avons choisie est développée tout au long de la deuxième partie.

Le langage objet des méthodes traductives est défini à la fin du chapitre 2.

#### 1.4. Définition de $f$

Nous avons déjà abordé la définition de cette fonction de correspondance entre les ensembles  $E$  et  $S$ .

Toutes les méthodes que nous avons vues, suivent la même démarche : elles transforment un programme écrit dans  $E$  en un objet formulé dans  $S$ .

Toutes effectuent une sorte de traduction dans un ensemble d'arrivée qui leur permet de se placer dans un système qui leur est propre et, par là, de

mettre en évidence différents aspects d'un programme :

- . la fonction mathématique qu'il dénote, indépendamment de la formulation de l'algorithme, pour les méthodes dénotationnelles,
- . le calcul qu'il réalise, pour les méthodes opérationnelles,
- . les propriétés des résultats d'un calcul, exprimées dans un système logique, pour les méthodes déductives,
- . la formulation en lambda-expression de la fonction qu'il remplit,
- . le programme équivalent écrit dans un autre langage, pour les méthodes traductives.

Toutes ces approches ont des objectifs différents mais passent bien par une transformation d'un programme en un autre objet.

#### 1.4.1. Méthodes interprétatives.

Ces méthodes négligent les problèmes syntaxiques et ne considèrent, en général, qu'une syntaxe abstraite du langage, en supposant les problèmes d'identification et les problèmes contextuels résolus en amont par un mécanisme qui ne les concerne pas.

La définition de  $f$  qui s'écrit :

$$\text{programme} \xrightarrow{f} \text{fonctions} \left( \begin{array}{l} \text{état} \rightarrow \text{état} \\ \text{donnée} \rightarrow \text{résultat} \\ \text{prédicat} \rightarrow \text{prédicat} \end{array} \right)$$

est alors remplacée par la définition d'une fonction à 2 arguments :

$$\begin{array}{l} \text{programme} \\ \text{abstrait} \end{array} , \begin{array}{l} \text{état} \\ \text{donnée} \rightarrow \text{résultat} \\ \text{prédicat} \end{array} \begin{array}{l} \text{état} \\ \text{prédicat} \end{array}$$

qui met en évidence l'interprète dont nous avons parlé au § 1.3.2.

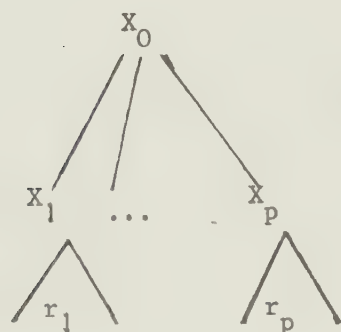
#### 1.4.2. Méthodes traductives.

Ces méthodes, au contraire, définissent directement  $f$ . Nous allons étudier, rapidement, cette définition pour des langages non contextuels, tout d'abord, puis pour des langages contextuels.

. langages non contextuels\*:

la fonction  $f$  peut être définie comme une fonction de ramifications [26] donnée récursivement.

A la règle de production  $X_0 \rightarrow X_1 \dots X_p$ , est associée la ramification suivante



notée  $X_0 \times (r_1 + \dots + r_p)$  où

$r_i$   $1 \leq i \leq p$  est une ramification étiquetée  $X_i$ .

On peut alors écrire

$$(1) \quad f(X_0 \times (r_1 + \dots + r_p)) = \phi_{X_0}(f(r_1), \dots, f(r_p))$$

où  $\phi_{X_0}$  est une fonction de traduction associée au symbole  $X_0$ .

On exprime ainsi que la traduction d'une ramification ne dépend que de la traduction des ramifications composantes et d'une fonction de traduction de la racine de la ramification. La localité de la traduction est mise en évidence, et la traduction est synthétisée depuis les feuilles vers la racine de la ramification.

. langages contextuels\*:

Ce modèle de définition est insuffisant pour les langages contextuels.

L'introduction du contexte peut se faire en prenant comme modèle :

---

\* nous appelons, ici, langages non contextuels, des langages dont l'ensemble  $E$  peut être complètement défini par une grammaire hors-contexte, et langages contextuels, des langages pour qui un modèle syntaxique hors-contexte est insuffisant mais, malgré tout, utilisé, les contraintes contextuelles étant formulées en dehors du modèle syntaxique. Nous ne suivons pas, ici la terminologie de la théorie des langages, à laquelle nous avons déjà dérogé, en notant  $E$  l'ensemble des programmes écrits, ensemble appelé langage dans cette théorie.

$$(2) \quad f(X_0 \times (r_1 + \dots + r_p), r) = \phi_{X_0}(f(r_1, r), \dots, f(r_p, r))$$

où  $r$  désigne la ramification complète du programme. La localité de la traduction est respectée mais le contexte englobant est connu.

Le chapitre 2 introduit à une définition de langage dans le style de la définition de ALGOL60. La fonction de traduction est du type (2) où le contexte global est transmis et supposé connu. Cette définition est complètement informelle car donnée en langue naturelle.

Le chapitre 5 introduit à la notion de double grammaire, utilisée dans la définition d'ALGOL68. Les contraintes contextuelles sont exprimées par une fonction du type (1), où les étiquettes des points de la ramification appartiennent à un ensemble infini, celui des notions, et contiennent l'information qui était contextuelle en ALGOL60. La définition est semi-formelle.

La première partie du chapitre 6 présente la méthode de Wirth-Weber. Cette méthode utilise un contexte manipulé par un algorithme décrit dans un certain langage. La définition doit être considérée comme formelle si le langage de description est supposé connu. La deuxième partie du chapitre 6 décrit la méthode des grammaires affixes. La définition de la traduction est dirigée par l'analyse syntaxique, algorithmiquement. Est donc définie une fonction récursive d'analyse qui possède un certain nombre de paramètres. Ceux-ci peuvent communiquer des informations sur la traduction pour accumuler la traduction générale comme paramètre de la racine de chaque ramification. Les informations transmises par paramètres sont de deux types : héritées, lorsqu'elles sont passées de la racine vers les feuilles, et synthétisées, des feuilles vers la racine.

Le chapitre 7, enfin, présente la méthode des attributs sémantiques qui généralise la notion de synthèse et d'héritage en conservant la propriété de localité.

Les fonctions récursives utilisées sont plus générales que (2). A chaque règle  $X_0 \rightarrow X_1 \dots X_p$ , est associé un certain nombre de fonctions qui définissent des concepts sémantiques distincts, dont fait partie la traduction.

On peut écrire ces fonctions comme :

$$a = \phi(a_1, \dots, a_n)$$

On définit ainsi l'attribut  $a$  associé au symbole  $X_i$  comme une fonction des attributs  $a_j$  des symboles  $X_k$  de la règle,  $\Phi$  dépendant de l'attribut  $a$  et de la règle considérée.

Enfin, le chapitre 8 compare les méthodes des chapitres 6 et 7 du point de vue de leur puissance descriptive et étudie le passage de l'une à l'autre.

## 2. DEFINITION DU TYPE ALGOL 60 [2]

### 2.1. Présentation de TURINGOL [1].

TURINGOL est un langage de programmation pour machines de Turing.

Rappelons qu'une machine de Turing est composée d'un ruban infini que nous supposerons être composé de cases, d'une tête de lecture-écriture et d'une unité de contrôle qui peut commander une des trois opérations suivantes :

- lecture du symbole situé dans la case actuellement sous la tête de lecture-écriture,
- écriture d'un symbole dans la case actuellement sous la tête de lecture-écriture,
- déplacement de la tête de lecture-écriture d'une case vers la gauche ou vers la droite.

Les caractéristiques de TURINGOL sont celles, simplifiées, d'un langage de programmation classique. Une déclaration annonce le vocabulaire des symboles de ruban. Des instructions permettent de tester le symbole courant par lecture implicite de la case courante, d'écrire un symbole dans la case courante, d'effectuer un déplacement gauche ou droit de la tête de lecture-écriture. Les notions d'instruction composée et d'instruction vide apparaissent naturellement pour aider à la programmation. Le flot de contrôle d'exécution est séquentiel et ne peut être rompu que par un test ou un branchement à une étiquette du programme.

### 2.2. Description formelle "à la ALGOL60".

Pour cette méthode de définition, l'ensemble des règles d'écriture est décrit dans un métalangage qui est la forme normale de Backus [3], ou BNF. La syntaxe du langage est donc définie par une grammaire hors contexte. L'ensemble S est décrit en langue naturelle, de façon informelle.

La fonction  $f$  de correspondance est décrite par simple juxtaposition de règles sémantiques aux règles d'écriture. Elle exprime, en langue naturelle, l'interprétation dans l'ensemble S des règles de E par une fonction récursive de synthèse d'information depuis les composants d'une règle d'écriture vers le sujet de cette règle. On doit noter que le contexte d'une règle est toujours supposé implicitement connu.

Les règles sémantiques jouent, ici, un double rôle :

- elles apportent des restrictions contextuelles qui n'ont pu être formulées par la grammaire hors contexte. Cet aspect de la sémantique que nous appellerons "sémantique statique" ([4] , [5]), n'est en fait que de la syntaxe déplacée par impossibilité de la formaliser dans le modèle syntaxique choisi.
- elles définissent, de façon informelle, l'effet ou la valeur de constructions syntaxiques lors de leur exécution. Cette "sémantique dynamique" traite toutes les informations qu'on ne peut extraire d'un programme à sa simple lecture car elles ne sont définies que dynamiquement. La signification d'un programme n'a alors de sens que lors de son exécution.

Dans les définitions sémantiques associées aux différentes règles de production, nous avons distingué les aspects statiques (S) des aspects dynamiques (D).

`<programme> ::= <déclaration> ; <liste d'instructions>.`

- . la déclaration permet de connaître le vocabulaire de ruban (S)
- . l'exécution d'un programme commence à la première instruction de la liste et se termine à la dernière (S-D)

`<déclaration> ::= TAPE ALPHABET IS <symbole>  
                  | <déclaration> , <symbole>`

- . tous les symboles déclarés doivent être distincts (S)

`<symbole> ::= id`

`<liste d'instructions> ::= <instruction>  
                          | <liste d'instructions> ; <instruction>`

- . les instructions d'une liste s'exécutent séquentiellement sauf lors de l'exécution d'une instruction `<goto>` ou éventuellement d'une instruction `<if>` (D). L'ordre séquentiel d'exécution est défini par l'ordre d'écriture des instructions (S).
- . après l'exécution de la dernière instruction de la liste, l'instruction qui suit la liste est exécutée (S-D)

`<instruction> ::= <étiquette> : <instruction>  
                  | <ins>`

- . une instruction, étiquetée ou non, a la même signification (S).
- . toutes les étiquettes d'un programme doivent être distinctes (S)
- . une étiquette repère et identifie l'instruction associée (S).

<étiquette> ::= id

<ins> ::= <print> | <if> | <vide> | <move> | <goto>  
| (<liste d'instructions>)

<print> ::= PRINT <symbole>

- . le symbole doit avoir été déclaré comme tel (S)
- . il remplace le symbole de la case courante (D)

<if> ::= IF THE TAPE SYMBOL IS <symbole>  
THEN <instruction>

- . le symbole doit avoir été déclaré comme tel (S)
- . il est comparé au symbole courant ; s'il y a coïncidence, l'instruction qui suit THEN est exécutée, sinon l'exécution de l'instruction if est terminée (S-D)

<vide> ::=

- . l'exécution de cette instruction est sans effet (S-D)

<move> ::= MOVE <sens> ONE SQUARE

- . la tête de lecture-écriture est déplacée à gauche ou à droite d'une case selon le sens (S-D)

<sens> ::= LEFT | RIGHT

- . LEFT signifie gauche, RIGHT droit (S)

<goto> ::= GOTO <étiquette>

- . l'étiquette doit avoir été déclarée comme telle (S)
- . la séquence est interrompue et reprend à l'instruction repérée par l'étiquette (S-D)

#### Remarques générales :

- . les symboles non terminaux sont notés entre chevrons
- . l'axiome est <programme>
- . le symbole terminal id est utilisé pour simplifier l'écriture, il correspond aux identificateurs ALGOL60.

## 2.3. Critiques de la méthode.

### 2.3.1. *Critiques des théoriciens.*

Une telle définition est peu satisfaisante car, à la rigueur de la notation syntaxique s'oppose l'imprécision et la verbosité de la sémantique. De plus, rien ne permet d'affirmer que des lacunes ne subsistent pas [6]. Dans notre définition, par exemple, on peut noter l'imprécision de la définition : "après l'exécution de la dernière instruction d'une liste, l'instruction qui suit la liste est exécutée".

Encore faudrait-il préciser ce qu'il convient de faire si la liste est elle-même la dernière instruction d'une liste ou la dernière instruction d'un programme.

### 2.3.2. *Critiques des usagers.*

Peut-on dire qu'un programme dans lequel une instruction non étiquetée suit une instruction de branchement est acceptable ? Oui, si on s'arrête à la définition non, dans l'esprit, car une définition qui ne signale pas cette erreur (de programmation ou de logique) ne remplit pas son rôle auprès de l'utilisateur. La même remarque peut être faite si dans une case apparaît un symbole non déclaré: ceci n'est pas considéré comme une erreur par la définition. On peut également citer l'exemple célèbre :

E : GOTO E ;

qui n'est pas traité comme une erreur.

### 2.3.3. *Critiques des implanteurs.*

L'imprécision est une catastrophe pour les implanteurs. La définition d'un langage est le législatif auquel ils doivent totalement obéir dans leur rôle exécutif. Ils connaissent bien la curieuse façon de procéder en cas d'imprécision du législatif, la jurisprudence, qui consiste à faire ce que leurs prédécesseurs ont fait ! Ce souci de compatibilité a même conduit [7] à considérer qu'un traducteur écrit pour un calculateur donné constitue la définition complète d'un langage.

Cette situation n'est pas supportable et a conduit à des modifications profondes dans la rédaction des normes de définition des langages, par exemple la future norme PL/1 [8].

## 2.4. Intérêt de la méthode.

Malgré toutes ces critiques, il est indiscutable que des définitions "à la ALGOL60" ont eu un grand succès, auprès des usagers que l'expression de la sémantique en langue naturelle ne rebute pas, bien au contraire, et auprès des implanteurs qui ont trouvé dans la définition syntaxique en BNF un outil privilégié de construction de traducteur.

Le passage de la définition synthétique de la syntaxe d'un langage en son expression analytique est simple dans la plupart des cas. La charpente rigoureuse que constitue un analyseur syntaxique obtenu automatiquement est la base de la construction de tous les traducteurs dirigés par la syntaxe. Les règles syntaxiques, enrichies par des "actions sémantiques" établies ponctuellement et effectuées lors de la reconnaissance des structures syntaxiques concernées, permettent simplement de traiter toute la sémantique statique.

Ces méthodes seront étudiées aux chapitres 6 et 7.

## 2.5. Définition de TL/1 [1]

### 2.5.1. *Présentation.*

Nous introduisons, ici, un autre langage, TL/1, qui est le langage objet des méthodes traductives des chapitres 6 et 7. La fonction de correspondance,  $f$ , permet, pour ces méthodes, de passer d'un programme Turingol au programme TL/1 correspondant. L'ensemble  $S$  est, dans ce cas, constitué des programmes TL/1 et on a déplacé le problème de la définition de  $S$  à celui de la définition de TL/1.

TL/1 possède 5 types d'instructions, toutes étiquetées par un nombre entier (ci-dessous,  $n_1$ )

|                    |                                                                                                                    |
|--------------------|--------------------------------------------------------------------------------------------------------------------|
| $n_1$ print $i$    | écrire, sur le ruban, le $i^e$ symbole déclaré,                                                                    |
| $n_1$ jump $n_2$   | branchement à l'instruction de numéro $n_2$                                                                        |
| $n_1$ if $i$ $n_2$ | si le symbole courant coïncide avec le $i^e$ symbole déclaré, la prochaine instruction est $n_1 + 1$ , sinon $n_2$ |
| $n_1$ move 1 ou -1 | déplacer la tête de lecture-écriture d'une case à gauche ou à droite                                               |
| $n_1$ stop         | fin d'exécution.                                                                                                   |

### 2.5.2. *Définition d'un interprète de TL/1.*

Pour définir la sémantique de TURINGOL, il suffit de définir TL/1. On pourrait imaginer de mettre en oeuvre une méthode de définition complète. Mais, nous allons nous limiter à en donner une définition interprétative sous forme d'un algorithme écrit en EULER [9].

Cet interprète peut être considéré comme une version simplifiée des interprètes des méthodes opérationnelles (chapitre 3).

L'environnement est constitué par

- le ruban                           noté T
- l'index du ruban               noté i
- le compteur ordinal           noté CO
- le programme TL/1           noté P

Pour éviter une formalisation inutile ici, on admet que la description des instructions TL/1 du paragraphe précédent constitue la définition de la syntaxe du langage.

```

CO ← 0 ; i ← -1 ;    /* état initial */
CYCLE : CO ← CO + 1 ; /* numéro de la prochaine instruction */
NEXT  : goto P [CO][1] ; /* aiguillage selon le type */
PRINT : T [i] ← P [CO][2] ; /* écriture du symbole */
      goto CYCLE ;
JUMP  : CO ← P [CO][2] ; /* sélection du numéro instruction */
      goto NEXT ;
IF    : if T [i] = P [CO][2] then goto CYCLE ;
      CO ← P [CO][3] ; /* sélection du numéro instruction */
      goto NEXT ;
MOVE  : i ← i + P [CO][2] ; /* modification selon sens */
      goto CYCLE ;
STOP  : stop ;

```

### 3. LANGAGE DE DEFINITION DE VIENNE (VDL) ([10] , [11], [12])

#### 3.1. Introduction.

VDL a été inspiré par les travaux de Mc Carthy sur LISP ([13], [14]).

Nous allons, pour cette méthode, définir les ensembles E, S et la fonction f introduits au chapitre 1.

L'ensemble des règles d'écriture est définie, en VDL, par deux syntaxes différentes. L'une donne la forme d'écriture concrète des programmes, l'autre donne une forme abstraite où ne subsistent que les informations importantes de chaque instruction, indépendamment du "sucre" syntaxique. Une fonction de transformation assure la correspondance entre les deux ensembles. On a

$$E_c \xrightarrow{\text{tr}} E_a$$

La représentation abstraite d'un programme est traitée par un interprète qui est l'abstraction d'un interprète informatique classique : tous les composants (contrôle, mémoire, instructions) de l'interprète, ont une représentation abstraite. Tous ces composants constituent un état de l'interprète.

L'ensemble S des valeurs sémantiques du langage est alors défini comme un ensemble de fonctions de transition d'un état dans un autre état.

VDL est un langage de définition des structures abstraites mises en oeuvre au cours du traitement. Son originalité est que le même langage sert à spécifier la représentation abstraite des programmes et des composants de l'interprète. On a ainsi une homogénéité que l'on ne trouve pas dans les autres approches interprétatives [13], [15].

#### 3.2. Syntaxe concrète, syntaxe abstraite.

##### 3.2.1. *Introduction.*

Cette transformation apparaît naturellement lorsqu'on regarde l'arbre syntaxique construit à partir d'une description en BNF, par exemple. On peut faire trois constatations sur un tel arbre :

- . il contient beaucoup d'informations inutiles comme certains symboles terminaux, certains noeuds intermédiaires,
- . il est ordonné par le texte source,
- . les noeuds sont nommés, les branches ne le sont pas.

L'objet élémentaire de VDL est la branche nommée par un sélecteur. Ce sélecteur est, à la fois, un nom et une fonction qui, appliquée à un objet possédant une branche nommée fournit l'objet accrochée à cette branche.

A partir de cet objet élémentaire, on peut alors construire des objets composés dont les branches sont nommées par des sélecteurs différents. On peut ainsi ne plus considérer un objet composé comme ordonné puisque chaque branche est désignable par son sélecteur.

Si on transforme un arbre syntaxique en ne conservant que les branches significatives et en les nommant, en supprimant les noeuds intermédiaires inutiles, on obtient un arbre sous forme abstraite normale [12]. Il faut alors définir l'ensemble des arbres abstraits par une nouvelle syntaxe abstraite, et l'exprimer en VDL. Pour cela, on introduit la notion de prédicat booléen qui, appliqué à un arbre abstrait, teste la nature de cet arbre en testant les objets sélectables.

### 3.2.2. Syntaxe abstraite de TURINGOL.

Les sélecteurs sont précédés du préfixe "s-", les prédicats du préfixe "is-".

is-programme = (<s-decl : is-decl>, <s-corps : is-ins-list>)

Le prédicat is-programme est vérifié par l'ensemble des objets composés de deux branches, l'une sélectable par s-decl conduit à un objet vérifiant is-decl, l'autre sélectable par s-corps conduit à un objet vérifiant is-ins-list. Les prédicats suffixés par "-list" sont vérifiés par des objets en nombre quelconque (n), sélectables par elem(i) pour  $1 \leq i \leq n$ , et conduisant à des objets vérifiant le prédicat non suffixé.

is-decl = is-symbol-list

is-ins = is-ins-e  $\vee$  is-ins-ne

is-ins-e = (<s-label : is-label-list>, <s-ins : is-ins-ne>)

is-symbol = is-id

is-label = is-id

is-ins-ne = is-goto-i  $\vee$  is-print-i  $\vee$  is-vide-i  $\vee$  is-cond-i  $\vee$   
is-comp-i  $\vee$  is-move-i

is-goto-i = (<s-type : GO> , <s-lab : is-label>)

is-print-i = (<s-type : PR> , <s-symbol : is-symbol>)

is-vide-i = <s-type : VIDE>

is-cond-i = (<s-type : COND> , <s-symbol : is-symbol>, <s-ins ; is-ins>)

is-comp-i = (<s-type : COMP>, <s-corps : is-ins-list>)

is-move-i = (<s-type : MV> , <s-sens : is-sens>)

is-sens = is-left  $\vee$  is-right

Nous avons supposé l'existence des prédicats élémentaires is-id, is-left, is-right.

Nous ne suivrons pas [12] dans la transformation texte concret-texte abstrait. Cette transformation peut être effectuée beaucoup plus simplement qu'en utilisant encore VDL comme langage de description de la transformation. On trouvera en annexe 1 la transformation par attributs sémantiques.

### 3.3. Etat de l'interprète.

Nous avons défini un état de l'interprète comme étant l'ensemble de ses composants. Ce sont :

- le programme sous forme abstraite,
- la mémoire de contrôle d'interprétation qui contient les instructions de l'interprète à exécuter et qui est organisée sous forme d'arbre,
- le ruban de la machine de Turing,
- un compteur ordinal donnant l'instruction à exécuter dans le programme,
- une librairie de sous-programmes interprétatifs.

Les sous-programmes sont de deux types :

- macro expansions :

$$\begin{array}{l} \text{nom}(t_1, \dots, t_n) = \\ \begin{array}{ccc} c_1 & \longrightarrow & t_1 \\ & \dots & \\ c_m & \longrightarrow & t_m \end{array} \end{array}$$

dans l'arbre de contrôle, la feuille "nom" est remplacée par l'arbre  $t_i$  qui correspond à la première condition  $c_i$  vraie.

- transformations de base

$$\begin{array}{l} \text{nom}(t_1, \dots, t_n) = \\ \text{PASS} : e_0 \\ a_1 : e_1 \\ \dots \\ a_m : e_m \end{array}$$

Les transformations de base peuvent modifier un quelconque composant de l'état, les expressions  $e_i$  étant affectées aux composants  $a_i$ , puis la valeur  $e_0$  est retournée au point d'appel. Un mélange de ces deux types de sous-programme est, bien entendu, autorisé.

A tout instant, l'instruction de l'interprète à exécuter est une des feuilles de l'arbre de contrôle. Lorsque l'arbre de contrôle est vide, l'interprétation est terminée.

L'état de l'interprète vérifie le prédicat :

is-état = (<s-prog : is-programme> , <s-lib : is-lib> ,  
           <s-c : is-control> , <s-tape : is-tape> ,  
           <s-ci : is-pile>)

is-tape = (<s-case : is-symbol-list> , <s-pos : is-entier>)

Le ruban d'entrée est considéré comme une liste de symboles, et un entier qui pointe sur la case courante dans la liste.

is-pile = (<s-s : is-entier> , <s-r : is-pile>)  $\forall \Omega$

Le compteur d'instruction est une pile dans laquelle sont mémorisés les rangs de diverses listes d'instructions, le sélecteur s-s donnant le rang dans la liste courante.

La description suivante de la librairie ne comprend pas l'instruction goto (sélecteur s-type = GO).

$\xi_0 = \mu_0$  (<s-prog : arbre abstrait> , <s-lib : librairie> ,  
           <s-c : int-prog (arbre abstrait)> ,  
           <s-tape : (<s-case : ruban> , <s-pos : -1>)> )  
           <s-ci : (<s-s : 1> , <s-r :  $\Omega$ >)> )

$\xi_0$  est l'état initial, l'état courant est noté  $\xi$

$\mu_0$  est l'opérateur de construction d'arbres.

### 3.4. Description de la librairie.

. int-prog (t) = int-linst(s-corps(t))

. int-linst(t) =

    s-s(s-ci( $\xi$ ))  $\leq$  length(t)  $\rightarrow$  int-linst(t)

        next

            int-inst(elem(s-s(s-ci( $\xi$ ))))(t)

vrai

$\rightarrow \Omega$

$\Omega$  représente l'objet vide.

La fonction length donne le nombre d'éléments d'une liste. La signification de la macro expansion int-linst est la suivante : si on n'a pas épuisé la liste d'instruction, remplacer dans l'arbre de contrôle, int-linst par int-linst next int-inst, donc, seront exécutées dans l'ordre

    int-inst            instruction courante

    puis    next            passer à l'instruction suivante .

    puis    int-linst

jusqu'à épuisement de la liste où int-linst est remplacé par l'objet vide.

```

. next =
    s-s(s-ci(ξ)) : s-s(s-ci(ξ)) + 1

. int-inst(t) =
    is-ins-e(t) → int-i(s-inst(t))
    vrai      → int-i(t)

. int-i(t) =
    is-print-i(t) → int-print(s-symbol(t))
    is-move-i(t)  → int-move(s)
                      s:sens(s-sens(t))
    is-vide-i(t)  → Ω
    is-cond-i(t)  → int-cond(s-symbol(t) , s-ins(t))
    is-comp-i(t)  → int-comp(s-corps(t))

. sens(t) =
    is-right(t) → PASS :-1
    is-left(t)  → PASS :+1

. int-print(s) =
    elem(s-pos(s-tape(ξ)))(s-case(s-tape(ξ))) : s

. int-move(i) =
    s-pos(s-tape(ξ)) : s-pos(s-tape(ξ)) + i

. int-cond(s,t) =
    elem(s-pos(s-tape(ξ)))(s-case(s-tape(ξ))) : s → int-inst(t)
    vrai → Ω

. int-comp(t) =
    vrai → pop
          int-linst(t)
          push

. push =
    s-ci(ξ) : μ₀(<s-s = 1> , <s-r = s-ci(ξ)>)

. pop
    s-ci(ξ) : s-r(s-ci(ξ))

```

### 3.5. Critiques de la méthode.

Signalons qu'un interprète complet traitant l'instruction goto figure en annexe 2. Nous demandons instamment aux personnes qui ne sont pas convaincues des problèmes que posent cette instruction de lire cette annexe et de la comparer à la définition précédente.

Nous avons déjà signalé, dans l'introduction, qu'une instruction qui cause de si profondes perturbations dans la définition formelle d'un langage peut être soupçonnée de nocivité inhérente et doit donc être manipulée avec beaucoup de soin. Mais, de par sa nature algorithmique, VDL permet la description d'une telle instruction. Nous verrons au chapitre 4 que certains modèles de définition sont plus restrictifs ou insuffisants et n'admettent pas cette instruction.

VDL remplit-il le triple but de satisfaire les théoriciens, les utilisateurs et les planteurs ?

#### 3.5.1. *Critiques des théoriciens.*

Pour les théoriciens, la réponse est "presque". Une étude [16] montre que VDL est de même puissance descriptive que d'autres modèles de définition dont celui du chapitre suivant. Nous soupçonnons cette étude de partialité car elle ne s'applique qu'à un langage très restreint et nous verrons que TURINGOL pose des problèmes à la méthode suivante.

Une critique des théoriciens concerne l'explicitation de tous les états intermédiaires à partir de l'état initial. Une critique de même nature est que l'interprète est défini de façon impérative, par opposition à une formulation déclarative, c'est-à-dire qu'il faille suivre explicitement toutes les transitions entre états par le déroulement dynamique d'un algorithme.

Une critique plus vive et justifiée tient au fait qu'on voit mal comment utiliser une telle définition pour établir la démonstration des propriétés du langage défini ou de son implantation. En particulier, on peut difficilement envisager de choisir VDL pour comparer les possibilités de deux langages.

#### 3.5.2. *Critiques des usagers.*

Pour les usagers, VDL est de peu d'intérêt par la complexité des mécanismes mis en oeuvre. On imagine mal un usager de PL/1 se penchant sur la définition formelle du langage : le vertige le guette. Mais les théoriciens répondraient, à juste titre, que la définition d'un langage conçu en dehors de méthodes de définition formelle ne peut conduire qu'à des descriptions extravagantes.

### 3.5.3. Critiques des implanteurs.

Pour les implanteurs, VDL est d'un gros intérêt car la méthodologie et les objets manipulés sont d'esprit typiquement informatique. L'interprète de VDL n'est que l'abstraction de calculateurs existant et, à ce titre, fournit une réponse à toutes les questions qui peuvent se poser au cours d'une implantation.

Il n'est pas question ici d'interprétations personnelles : la machine, c'est-à-dire, l'interprète, est là et donne la réponse à quelque question que l'on se pose. VDL est donc un manuel de référence complet d'un langage. Une variante de VDL a d'ailleurs été choisie comme méthode de définition de la future norme PL/1 [8]. Mais il est dommage qu'un tel document ne soit une bible que pour les implanteurs et pas pour les usagers.

Une critique vigoureuse que font les implanteurs orientés vers la traduction et non l'interprétation, est que VDL n'apporte qu'une explication complète de ce qui doit se passer au cours de l'exécution d'un programme, mais ne laisse aucun espoir d'obtention automatique de traducteurs. Ceci est une évidence, VDL ne constitue qu'une description à l'aide d'une machine qui, bien qu'étant l'abstraction de machines informatiques n'est pas moins extrêmement éloignée des calculateurs commerciaux.

Un autre exemple d'approche interprétative, mais orientée vers l'obtention automatique d'interpréteurs plus efficaces, est donné par COUSOT [17].

## 4. DEFINITION AXIOMATIQUE OU DEDUCTIVE [18]

### 4.1. Introduction

Cette méthode considère que la valeur sémantique d'un programme est une relation de déduction entre propriétés des données et propriétés des résultats [21].

La sémantique d'une instruction est définie par des règles de déduction qui permettent de démontrer un théorème sur les nouvelles valeurs des données après l'exécution de cette instruction, à partir des propriétés vérifiées avant l'exécution. Pour un programme, on obtient ainsi, instruction par instruction, une suite de théorèmes jusqu'à un théorème final sur la fonction réalisée par le programme. Une preuve de correction peut donc être effectuée, à l'aide de la méthode, en prouvant qu'une assertion sur la fonction réalisée à la fin du programme, peut se déduire des théorèmes obtenus sur les données. Nous donnerons un exemple de preuve au § 4.4.

Si on considère la définition du langage Pascal [64], on peut s'étonner de l'absence de l'instruction goto. Une raison essentielle est que cette instruction rompt la structure normale du déroulement séquentiel des programmes et que la méthode qui est basée sur un établissement successif des propriétés des données ne se prête que très mal à la description d'une telle instruction.

En effet, la séquentialité des instructions permet de définir simplement l'effet d'une instruction en fonction de sa précédente lexicographique qui est aussi sa précédente en exécution. Or, une instruction de branchement rompt cette structure harmonieuse mais simpliste. Ainsi, on devrait être capable de décrire que deux instructions écrites en séquence mais séparées par une étiquette établissent sur les données les mêmes propriétés que toutes les instructions qui précèdent une instruction de branchement à cette étiquette.

Un essai [65] a été fait de définir un langage avec goto mais de très sévères restrictions ont été apportées qui rendent cette instruction quasi inutilisable.

Ce point soulève un problème sérieux : dans un tel cas, est-ce la méthode de définition qui est mal adaptée car insuffisante ou est-ce le concept, qui n'est pas descriptible, qui doit être exclu du langage ? Cette question est au coeur de la polémique sur la programmation structurée et nous nous garderons bien de prendre parti. On doit cependant remarquer qu'il serait étonnant qu'il n'y ait qu'un coupable ! Il est probable qu'un concept, qui trouble à ce point une méthode de définition formelle alors

que la plupart des concepts usuels des langages peuvent être décrits simplement, est à éviter. Mais on ne saurait admettre d'aller jusqu'à sa suppression car se serait s'engager sur une pente dangereuse : la mise en place de critères idéaux de définition de langages, par exclusion de toute perturbation.

Le langage TURINGOL contient (malheureusement ou heureusement ?) une instruction de branchement, et, qui plus est, il autorise des branchements à toute instruction d'un programme : au milieu d'une liste d'instructions aussi bien qu'à une instruction conditionnée par un "if".

Dans ces conditions, nous pensons qu'il est impossible de définir le langage par cette méthode et nous allons, afin de donner un exemple de preuve de correction, modifier TURINGOL en interdisant l'instruction goto (la notion d'étiquette devient alors sans objet), et en introduisant une instruction d'itération.

La définition de cette instruction est faite, ici, dans le style d'ALGOL60 (chapitre 2) puis par la méthode axiomatique.

```
<instruction> ::= WHILE THE TAPE SYMBOL IS <symbole>
                  DO      <instruction>
```

- le symbole doit avoir été déclaré comme tel,
- l'instruction est exécutée tant que le symbole coïncide avec le symbole courant.

#### 4.2. Présentation de la méthode.

Les instructions du langage sont notées S. Des formules logiques, notées P, Q et R, sont établies pour décrire les propriétés des données du langage. Des assertions, notées H, ainsi que des règles de déduction sont définies afin d'axiomatiser les instructions du langage.

Les assertions sur les instructions s'écrivent :

$$P \{S\} Q$$

et se lisent : si P est vrai avant l'exécution de S et si S se termine, alors Q est vraie après l'exécution de S.

Les règles de déduction s'écrivent

$$\frac{H_1, H_2, \dots, H_n}{H}$$

et se lisent : si H<sub>1</sub>, ..., H<sub>n</sub> sont des assertions vérifiées, alors H est une assertion vérifiée.

### 4.3. Définition de TURINGOL - bis.

La seule donnée, que peut manipuler un programme, est le ruban de lecture-écriture, dont le vocabulaire est introduit par une déclaration.

. vocabulaire de ruban  $V$  :

TAPE ALPHABET IS  $s_1, s_2, \dots, s_n$

-  $s_1, s_2, \dots, s_n$  sont des éléments distincts de  $V$

- ce sont les seuls éléments de  $V$ .

. ruban

Le ruban est composé d'un ensemble de cases prenant leur valeur dans  $V$ . Chaque case est identifiée par son rang que l'on peut considérer comme une fonction de l'ensemble des entiers  $N$  dans l'ensemble  $V$

$$r : N \rightarrow V$$

La case courante est identifiée par un entier noté  $i$ , dont la valeur initiale est fixée à  $-1$ .

Le ruban est noté  $\{\dots r_2 r_1 r_0 r_{-1} r_{-2} \dots, i\}$ , la case courante est notée  $r_i$ .

Nous introduisons une deuxième fonction  $\rho$

$$\rho : N \rightarrow V$$

qui donne le contenu initial des cases du ruban.

On a donc, initialement,

$$\forall k \in N \quad r_k = \rho_k$$

. instruction PRINT  $P_s^i \{\text{PRINT } s\} P$

La notation  $P_y^x$  est utilisée pour représenter le remplacement de  $x$  par  $y$  dans la formule logique  $P$ .

. Instruction MOVE  $P_{i-1}^i \{\text{MOVE RIGHT ONE SQUARE}\} P$

$P_{i+1}^i \{\text{MOVE LEFT ONE SQUARE}\} P$

. instruction vide  $P \{\text{vide}\} P$

|                        |                                                                                |
|------------------------|--------------------------------------------------------------------------------|
| . instruction IF       | $P \wedge r_i = s \{S\} Q, P \wedge r_i \neq s \supset Q$                      |
|                        | <hr/>                                                                          |
|                        | $P \{ \text{IF THE TAPE SYMBOL IS } s \text{ THEN } S \} Q$                    |
| . liste d'instructions | $P \{S\} Q, Q \{S'\} R$                                                        |
|                        | <hr/>                                                                          |
|                        | $P \{S ; S'\} R$                                                               |
| . instruction WHILE    | $P \wedge r_i = s \{S\} P$                                                     |
|                        | <hr/>                                                                          |
|                        | $P \{ \text{WHILE THE TAPE SYMBOL IS } s \text{ DO } S \} P \wedge r_i \neq s$ |

#### 4.4. Preuve de correction d'un programme.

Soit le programme TURINGOL-bis suivant :

```
TAPE ALPHABET IS UN, ZERO ;
MOVE LEFT ONE SQUARE ;
WHILE THE TAPE SYMBOL IS UN DO
    ( PRINT ZERO ;
      MOVE LEFT ONE SQUARE ;
    ) ;
PRINT UN ;
```

Supposons que le ruban d'entrée contienne, initialement, un nombre binaire<sup>\*</sup> et que le chiffre de droite de ce nombre soit une case à gauche de la position de la tête de lecture-écriture.

Nous voulons prouver que le programme précédent ajoute 1 au nombre binaire initial.

De la déclaration, on déduit :

$$V = \{UN, ZERO\}$$

et la proposition initiale

$$P_1 = [i = -1 \text{ et } \forall k \in \mathbb{N} \quad r_k = \rho_k]$$

MOVE LEFT ONE SQUARE ;

donne  $P_2 [i = 0 \text{ et } \forall k \in \mathbb{N} \quad r_k = \rho_k]$

Cette proposition est valide à l'entrée dans la boucle WHILE.

---

\*Par commodité d'écriture, nous utiliserons, pour la numération binaire, les symboles UN et ZERO.

L'assertion générale que nous utiliserons à l'entrée de la boucle, c'est-à-dire, au niveau du test du symbole courant est :

$$H = [\forall j \quad 0 \leq j < i \quad r_j = 0 \wedge \rho_j = 1 \\ \text{et } \forall k \quad i \leq k \quad r_k = \rho_k]$$

Pour  $i = 0$   $P_2 \supset H$ ,  $H$  est démontrée.

WHILE THE TAPE SYMBOL IS UN DO

$$[\forall j \quad 0 \leq j < i \quad r_j = 0 \wedge \rho_j = 1 \\ \text{et } \forall k \quad i \leq k \quad r_k = \rho_k \text{ et } r_i = 1] \equiv$$

$$\forall j \quad 0 \leq j < i \quad r_j = 0 \wedge \rho_j = 1 \\ \text{et } \forall k \quad i < k \quad r_k = \rho_k \text{ et } r_i = 1 \text{ et } \rho_i = 1]$$

( PRINT ZERO ;

$$[\forall j \quad 0 \leq j < i \quad r_j = 0 \wedge \rho_j = 1 \\ \text{et } \forall k \quad i \leq k \quad r_k = \rho_k \text{ et } r_i = 0 \text{ et } \rho_i = 1] \equiv \\ \forall j \quad 0 \leq j \leq i \quad r_j = 0 \wedge \rho_j = 1 \\ \text{et } \forall k \quad i < k \quad r_k = \rho_k]$$

MOVE LEFT ONE SQUARE ;

$$[\forall j \quad 0 \leq j < i \quad r_j = 0 \wedge \rho_j = 1 \\ \text{et } \forall k \quad i \leq k \quad r_k = \rho_k]$$

Ce théorème est bien identique à l'assertion d'entrée de boucle  
) ;

Dans le cas de sortie de boucle :

$$[\forall j \quad 0 \leq j < i \quad r_j = 0 \wedge \rho_j = 1 \\ \text{et } \forall k \quad i \leq k \quad r_k = \rho_k \text{ et } r_i \neq 1] \equiv \\ [\forall j \quad 0 \leq j < i \quad r_j = 0 \wedge \rho_j = 1 \\ \text{et } \forall k \quad i < k \quad r_k = \rho_k \text{ et } r_i = 0 \text{ et } \rho_i = 0]$$

PRINT UN ;

$$[\forall j \quad 0 \leq j < i \quad r_j = 0 \wedge \rho_j = 1 \\ \text{et } \forall k \quad i < k \quad r_k = \rho_k \text{ et } r_i = 1 \text{ et } \rho_i = 0]$$

qui est le théorème final.

Les valeurs initiales et finales du contenu du ruban peuvent être calculées à l'aide de ce théorème.

$$\text{Valeur initiale} = \rho_0 + 2\rho_1 + \dots + 2^{i-1}\rho_{i-1} + 2^i\rho_i + a$$

$$\text{Théorème } \supset \quad = 1 + 2 + \dots + 2^{i-1} + 0 + a$$

$$\text{Valeur finale} = r_0 + 2r_1 + \dots + 2^{i-1}r_{i-1} + 2^ir_i + a$$

$$\text{Théorème } \supset \quad = 0 + 2^i + a$$

$$\begin{aligned} \text{Valeur finale} - \text{Valeur initiale} &= 2^i - (1+2+\dots+2^{i-1}) \\ &= 2^i - (2^i-1) \\ &= 1 \end{aligned}$$

Le programme a ajouté 1 au nombre binaire initial.

Ce résultat n'est garanti par la méthode que si l'algorithme se termine. Une preuve de terminaison est de trouver une quantité positive qui décroît à chaque boucle. Si le nombre binaire initial avait  $n$  chiffres binaires,  $n-i$  est une telle quantité.

#### 4.5. Critiques de la méthode.

Nous avons déjà fait une critique lorsque nous n'avons pas pu définir TURINGOL avec cette méthode. Cette critique correspond au fait que cette méthode s'applique à des langages "idéaux". Un autre exemple est l'écriture  $P_e^v \{v := e\} P$ , pour une affectation de  $e$  à  $v$  qui sous entend l'interdiction d'effets de bord dans l'expression  $e$  puisque seule la variable  $v$  est modifiée. Dans la description de Pascal [64], l'appel d'une fonction sous entend qu'aucune variable extérieure à la fonction n'est modifiée. Un dernier exemple montre l'ignorance de contraintes informatiques importantes : la représentation des données en mémoire, qui est nécessairement finie pour un informaticien, alors que des axiomes mathématiques ne peuvent pas toujours exprimer cette limitation.

Tous ces exemples montrent les difficultés de transposer dans une théorie logique, pour cette méthode (mais les autres méthodes mathématiques ont des difficultés analogues) des concepts informatiques. Mais ces méthodes ont eu l'immense mérite de mettre en évidence les défauts les plus graves des langages actuels et ont entraîné une nouvelle génération de langages qui, bâtis à l'aide de modèles formels, offrent de plus des avantages de sécurité accrue pour les usagers. On peut citer un exemple convaincant, le langage LIS [22].

Mais il n'empêche que les langages actuels existent et il serait fâcheux qu'on ne s'intéresse pas à eux s'il n'ont pas le bon goût d'entrer dans le cadre d'un modèle de définition peut-être trop restreint.

Si nous sommes d'accord pour une sensibilisation à des inconvénients, nous ne pouvons admettre une démission : après l'anathème jeté aux instructions de branchement, une réhabilitation partielle de ces instructions est en cours, ce qui semble, informatiquement, plus raisonnable.

Cette méthode satisfait-elle les théoriciens ?

Curieusement, la réponse est non. Les reproches qui lui sont faits sont une grande imprécision dans les notations, en particulier sur la nature des formules logiques autorisées, ce qui est gênant lorsque l'utilisateur doit introduire ses assertions et donc les formuler ; une trop large part est laissée à l'intuition ; aucune preuve de consistance n'a été faite pour la méthode et donc, rien ne prouve qu'on ne peut pas démontrer une propriété et la propriété inverse.

En fait, il semble que cette méthode est surtout destinée aux utilisateurs qui, à défaut de preuve de correction, peuvent en espérer une conviction de correction de leurs programmes.

Mais il faut dire que cette conviction n'est pas toujours facile à obtenir et que, déjà sur notre exemple, l'accumulation des propriétés des données est importante alors que TURINGOL est très pauvre sur l'aspect données manipulées. On peut imaginer les proportions que prendraient les théorèmes obtenus pour un programme important manipulant des données complexes.

Une dernière critique vient des planteurs qui ne voient, malheureusement, pas de possibilité de déduction automatique d'implantation à partir de ces méthodes. Nous avons déjà souligné l'espoir de voir la complémentarité de toutes les méthodes de définition s'étendre afin d'offrir un éventail élargi de modèles de définitions, adaptés à l'usage que l'on veut en faire, sans oublier les méthodes traductives.

## 5. DEFINITION GRAMMATICALE DE VAN WIJNGAARDEN [23]

5.1. Introduction

Nous avons souligné au § 1.2, que la sémantique statique d'un langage n'est, en fait, que de la syntaxe déplacée par impossibilité de la formaliser dans une grammaire hors contexte.

Les grammaires de VAN WIJNGAARDEN ont été introduites pour permettre le traitement syntaxique de dépendances contextuelles. Elles sont connues sous le nom de "doubles grammaires" ou "W-grammaires". Ce nom de doubles grammaires vient de l'idée de base qui est qu'un premier niveau de grammaire va, s'appliquant à un deuxième niveau, engendrer la grammaire finale décrivant le langage. Dans ce chapitre, nous ne nous intéresserons qu'à la définition syntaxique du langage, c'est-à-dire à l'ensemble E.

5.2. Définitions5.2.1. *Grammaire de premier niveau.*

Cette grammaire est une grammaire hors contexte classique, mais sans définition d'axiome :

$$G(m) = \{M, S^+, R, m\} \quad m \in M$$

où M est un ensemble fini de métanotations que nous noterons en lettres majuscules,

S est l'alphabet des lettres minuscules. Les éléments de  $S^+$  sont appelés protonotations,

R est un sous ensemble fini de  $M \times (M \cup S^+)^*$

dont les éléments sont appelés méta-règles et notés  $x :: y_1 y_2 \dots y_n$ .

si  $x :: w_1$  et  $x :: w_2$ , on peut écrire  $x :: w_1 ; w_2$ .

5.2.2. *Grammaire de deuxième niveau.*

Cette grammaire n'est pas une grammaire hors contexte. Elle est définie par

$$\{M, S^+, T, Z, P\}$$

où M est l'ensemble des métanotations de la grammaire du premier niveau,

$S^+$  est l'ensemble des protonotations

T est un ensemble fini de "symboles" notés en lettres minuscules et se terminant par "symbol".

Ce sont les symboles terminaux de la grammaire.

$Z$  est un élément de  $S^+$ , c'est l'axiome de la grammaire

$P$  est un sous ensemble fini de  $(S^+ \cup M)^+ \times ((S^+ \cup M)^+ \cup T)^*$  dont les éléments sont appelés "hyper-règles" et notés

$$x : y_1, y_2, \dots, y_n.$$

si  $x : w_1$  et  $x : w_2$ , on peut écrire  $x : w_1 ; w_2$ .

### 5.2.3. Terminologie.

Une métanotion est un élément de  $M$

Une protonotion est un élément de  $S^+$

Une hypernotation est un élément de  $(S^+ \cup M)^+$

Soit  $m \in M$ . Par application de la grammaire de premier niveau  $G(m)$ , on obtient l'ensemble  $PT(m)$  des productions terminales de  $m$ .

Une caractéristique importante des hyper-règles est le "remplacement uniforme" d'une métanotion (noté RU).

Considérons une hyper-règle : toutes les occurrences de la même métanotion doivent être uniformément remplacées par une même production terminale.

Ceci permet de définir une notion qui est toute protonotion obtenue par remplacement uniforme des métanotions intervenant dans une hypernotation.

### 5.2.4. Commentaires.

Ces définitions ne sont pas évidentes à saisir. Quelques explications informelles aideront à les comprendre :

La grammaire du premier niveau introduit toutes les entités nécessaires aux contraintes contextuelles. La grammaire du deuxième niveau peut être considérée comme un modèle grammatical dans lequel il convient d'appliquer la règle du remplacement uniforme pour exprimer les contraintes contextuelles. On obtient ainsi une grammaire hors contexte, en général infinie, qui est la grammaire du langage.

### 5.2.5. Exemple 1.

Le premier exemple décrit le langage  $a^n b^n$   $n > 0$

#### Méta-règles

$N :: n ; Nn$ .

$AB :: a ; b$ .

#### Hyper-règles

$s : Na, Nb$ .

$nN AB : \text{lettre } AB \text{ symbol}, N AB$ .

$n AB : \text{lettre } AB \text{ symbol}$ .

axiome  $s$ .

Construction de la grammaire du langage :

dans l'hyper-règle  $n \ N \ AB$  : lettre  $AB$  symbol,  $N \ AB$  .

appliquons la règle de RU pour la métanotation  $AB$  . On obtient

$s$  :  $Na, Nb$  .

$n \ N \ a$  : lettre  $a$  symbol,  $N \ a$  .

$n \ N \ b$  : lettre  $b$  symbol,  $N \ b$  .

puis pour  $n \ A \ B$  : lettre  $A \ B$  symbol .

$n \ a$  : lettre  $a$  symbol

$n \ b$  : lettre  $b$  symbol

puis, en appliquant la règle de RU pour  $N$  , on obtient :

$s$  :  $na, nb$  ;  $nna, nnb$  ; ... ;  $n^p a \ n^p b$  ; ...

$na$  : lettre  $a$  symbol .

$nb$  : lettre  $b$  symbol .

$nna$  : lettre  $a$  symbol,  $na$  .

$nnb$  : lettre  $b$  symbol,  $nb$  .

...

$n^p a$  : lettre  $a$  symbol,  $n^{p-1} a$  .

$n^p b$  : lettre  $b$  symbol,  $n^{p-1} b$  .

...

Ceci constitue la grammaire (infinie) engendrée par la double grammaire.

#### 5.2.6. Exemple 2.

Le deuxième exemple va nous permettre d'introduire deux définitions supplémentaires : métanotation indicée et protonotation non définie ou impasse.

Lorsque, dans une hyper-règle apparaît plusieurs fois la même métanotation, la règle du RU ne s'applique qu'aux métanotations de même indice. Si par exemple, on a l'hyper-règle :

...  $A1$  ...  $A2$  ... : ...  $A2$  ...  $A1$  ...  $A2$

les méta-notions  $A1$  et  $A2$  doivent être considérées comme des occurrences distinctes de la méta-notion  $A$ . Ceci est équivalent à créer deux méta-règles supplémentaires :

$A1 :: A$  .

$A2 :: A$  .

Dans cet exemple, nous voulons décrire les chaînes de deux lettres où la première lettre précède alphabétiquement la deuxième.

méta-règles :

LET :: a ; b ; c ; ... ; z .

LETY :: EMPTY ; LETY, LET .

EMPTY :: .

hyper-règles :

s : lettre LET1 symbol, lettre LET2 symbol,

ou LET1 précède LET2 dans abcdefghijklmnopqrstuvwxyz

ou LET1 précède LET2 dans LETY1 LET1 LETY2 LET2 LETY3 : EMPTY

Cet exemple illustre la notion de prédicat et d'impasse.

Construisons la grammaire pour la chaîne b1 puis pour la chaîne lb.

- i) s : lettre b symbol, lettre l symbol,  
ou b précède l dans a b c ... z.  
ou b précède l dans a b c ... k l m ... z : .

Le membre gauche de la dernière production est une notion conduisant à la chaîne vide : c'est un prédicat vérifié.

- ii) s : lettre l symbol, lettre b symbol,  
ou l précède b dans a b c ... z.

La notion "ou l précède b dans a b c ...z" ne peut être construite, c'est un prédicat non vérifié ou une impasse.

Ce mécanisme de prédicat construit à partir d'une notion vide ou inexistante, est le mécanisme de base d'expression des contraintes contextuelles.

### 5.3. Définition des règles grammaticales de TURINGOL.

Dans un langage comme TURINGOL, les contraintes contextuelles concernent essentiellement l'identification des variables : il faut vérifier que les déclarations des symboles de ruban et les définitions d'étiquettes, sont correctes ainsi que les références aux uns et aux autres.

Une difficulté, dans TURINGOL est que, si les déclarations de symboles de ruban sont groupées en tête d'un programme, les définitions d'étiquettes sont dispersées dans le programme. Cet aspect entraîne une description quelquefois lourde.

## Méta-règles :

```

TAPE      :: DEFT ; TAPE DEFT .
TAPETY    :: TAPE ; EMPTY .
DEFT      :: tape ID .
LABEL     :: DEFL ; LABEL DEFL .
LABETY    :: LABEL ; EMPTY .
DEFL      :: label ID .
ID        :: L      ; ID   L .
L         :: lettre A .
A         :: a ; b ; ... ; z .
EMPTY     :: .
ALPHABET  :: abc ... z .
S         :: left ; right .
TYPE      :: tape ; label .
TALA      :: TAPE ; LABEL .
N         :: A      ; N A .
NETY      :: N      ; EMPTY .

```

## Commentaires :

- . les déclarations de ruban sont engendrées par la métanotion TAPE sous la forme tape lettre .. lettre .. tape lettre ...
- . les définitions d'étiquettes sont engendrées par la métanotion LABEL avec le préfixe label.
- . toutes les métanotions de suffixe ETY peuvent dériver la chaîne vide.

Hyper-règles :

programme : declare symbol, declaration de TAPE,  
                   TAPE <LABETY> <LABETY> instruction suite,  
                   ou LABETY unique.

declaration de DEFT TAPE : declaration de DEFT,  
                   virgule symbol, declaration de TAPE,  
                   ou DEFT n est pas dans TAPE .

declaration de TYPE ID : ID decl .

L ID decl : L symbol, ID decl .

L decl : L symbol .

TAPE <LABETY1> <LABETY2> instruction suite :  
                   LABETY3 TAPE <LABETY1> <LABETY4> instruction,  
                   ptvirg symbol, TAPE <LABETY1> <LABETY5> instruction suite,  
                   ou LABETY2 est LABETY3 LABETY4 LABETY5 ;  
                   LABETY6 TAPE <LABETY1> <LABETY7> instruction,  
                   ou LABETY2 est LABETY6 LABETY7.

LABETY1 TAPE <LABETY2> <LABETY3> instruction :  
                   definition de LABETY1,  
                   TAPE <LABETY2> <LABETY3> inst.

TAPE <LABETY1> <LABETY2> inst :  
                   TAPE écrire, ou LABETY2 est EMPTY ;  
                   vide, ou LABETY2 est EMPTY ;  
                   bouger, ou LABETY2 est EMPTY ;  
                   LABETY1 allera, ou LABETY2 est EMPTY ;  
                   TAPE <LABETY1> <LABETY2> tester ;  
                   TAPE <LABETY1> <LABETY2> composer .

definition de EMPTY : EMPTY .

definition de DEFL LABETY : declaration de DEFL,  
                   deux points symbol, definition de LABETY.

TAPE écrire : print symbol, ID decl, ou tape ID est dans TAPE .  
vide : EMPTY .

TAPE <LABETY1> <LABETY2> tester :  
    if symbol, ID decl, ou tape ID est dans TAPE,  
    then symbol,  
    LABETY3 TAPE <LABETY1> <LABETY4> instruction,  
    ou LABETY2 est LABETY3 LABETY4 .

bouger : move symbol, S symbol, un symbol.

LABETY allera : goto symbol, ID decl, ou label ID est dans LABETY .

TAPE <LABETY1> <LABETY2> composer :  
    parg symbol, TAPE <LABETY1> <LABETY2> instruction suite,  
    pard symbol .

Commentaires :

- . Les chevrons < et > servent à éviter les ambiguïtés.
- . La dispersion de la définition des étiquettes oblige à créer des métanotations LABETY en grand nombre. Ainsi dans l'hyper-règle définissant instruction suite

LABETY1 représente l'ensemble des étiquettes référençables,  
LABETY2 les étiquettes non encore déclarées,  
LABETY3 les étiquettes à déclarer pour la prochaine instruction,  
ou LABETY6

LABETY4 les étiquettes à déclarer dans le corps de la prochaine  
ou LABETY7 instruction (cas du if et de l'instruction composée),

LABETY5 les étiquettes à déclarer pour les instructions suivantes,  
avec des contraintes par prédicat sur ces ensembles éventuellement vides d'étiquettes.

- . La non duplication des définitions d'étiquettes est vérifiée par le prédicat "ou LABETY est unique".

Prédicats :

vrai : EMPTY .

ou DEFL LABETY unique :  
    ou DEFL n est pas dans LABETY,  
    ou LABETY unique.

ou EMPTY unique : vrai .  
 ou DEFL est dans LABETY1 DEFL LABETY2 : vrai .  
 ou DEFT est dans TAPETY1 DEFT TAPETY2 : vrai .  
 ou LABETY est LABETY : vrai .  
 ou label ID1 n est pas dans LABETY1 label ID2 LABETY2 :  
     ou ID1 diffère de ID2  
 ou DEFL n est pas dans EMPTY : vrai .  
 ou ID1 lettre L1 diffère de ID2 lettre L2 :  
     ou ID1 diffère de ID2 ;  
     ou L1 précède L2 dans ALPHABET ;  
     ou L2 précède L1 dans ALPHABET .  
 ou ID diffère de EMPTY : vrai .  
 ou EMPTY diffère de ID : vrai .  
 ou L1 précède L2 dans NETY1 L1 NETY2 L2 NETY3 : vrai .  
 ou tape ID1 n est pas dans TAPETY1 tape ID2 TAPETY2 :  
     ou ID1 diffère de ID2 .

Symboles terminaux :

|                    |                       |
|--------------------|-----------------------|
| declare symbol     | TAPE ALPHABET IS      |
| virgule symbol     | ,                     |
| lettre a symbol    | A                     |
| ...                | ...                   |
| lettre z symbol    | Z                     |
| ptvirg symbol      | ;                     |
| deux points symbol | :                     |
| print symbol       | PRINT                 |
| if symbol          | IF THE TAPE SYMBOL IS |
| goto symbol        | GOTO                  |
| move symbol        | MOVE                  |
| left symbol        | LEFT                  |
| right symbol       | RIGHT                 |
| un symbol          | ONE SQUARE            |
| parg symbol        | (                     |
| pard symbol        | )                     |

#### 5.4. Critiques de la méthode.

Une critique évidente de la méthode est sa lourdeur et sa complexité. Cette lourdeur se manifeste sur le plan synthétique qui intéresse les usagers et sur le plan analytique pour les planteurs.

- synthétique : les W-grammaires sont de manipulation difficile. Si l'on considère la définition d'Algol 68 [23], et si l'on s'intéresse à un point particulier du langage, par exemple l'identification des étiquettes, il est difficile d'extraire de la définition les hyper-règles qui traitent ce point. De plus, il est certainement hasardeux d'affirmer d'une définition que tous les concepts sont décrits conformément à ce qu'on en attendait.

Les W-grammaires sont un outil d'une puissance difficilement contrôlable qui permettent difficilement une définition modulaire d'un langage.

- analytique : la principale critique que l'on puisse faire est qu'il est problématique d'envisager l'obtention automatique d'un analyseur de W-grammaires. A notre connaissance, la seule implantation réalisée ([24]) comporte de sévères restrictions et ne permet que la reconnaissance de textes très courts. Cette difficulté tient au sens de parcours des informations contextuelles : il est nécessaire de les connaître toutes pour les répercuter, par la règle du remplacement uniforme, dans les notions de la grammaire.

Il est aisé d'hériter des informations depuis des notions globales vers des notions partielles mais la synthèse des informations dans le sens contraire n'est qu'implicite : par exemple, dans l'hyper-règle programme, considérons les deux occurrences de la méta-notion TAPE dont on est certain, par la règle du RU, qu'elles représentent les mêmes productions terminales. La première joue implicitement un rôle de synthèse, la seconde permet d'hériter la même information. Synthèse implicite et héritage limité par la règle du RU sont les critiques essentielles qu'on peut formuler.

Un point très positif doit être noté : cette méthode de description est déclarative, elle n'impose aucune contrainte du type synthèse puis héritage de gauche à droite. Notre définition de TURINGOL, par exemple, nécessite la synthèse de toutes les étiquettes avant tout remplacement uniforme et ne peut se faire, algorithmiquement, en un seul passe gauche droite sur le programme. Rien dans la définition, n'impose ces deux passes ; ce n'est que lors d'une mise en oeuvre effective que la nécessité des deux passages apparaît. Cet aspect déclaratif nous semble important car il laisse la méthode analytique de réalisation libre des choix d'implantation.

### 5.5. W-grammaires et sémantique.

Nous nous sommes limités jusqu'à maintenant à étudier les propriétés grammaticales des doubles grammaires et à montrer l'expression de la sémantique statique par cette méthode.

La sémantique dynamique d'un programme est exprimée, dans le rapport révisé ALGOL 68 [23] "par l'intermédiaire de phrases (dans une langue naturelle formalisée d'une certaine manière) qui spécifient les actions à entreprendre au cours de l'élaboration de ce programme". L'expression de cette sémantique ne diffère donc pas fondamentalement de la méthode du chapitre 2 : elle est décrite par juxtaposition aux règles syntaxiques, d'actions sur une machine hypothétique.

On peut cependant remarquer que des tentatives ont été faites pour utiliser le mécanisme des doubles grammaires pour la définition complète d'un langage [55]. Dans cet exemple, la définition du langage couvre les aspects syntaxiques en décrivant les programmes corrects mais aussi l'ensemble des entrées possibles accompagnées des sorties correspondantes. Pour cela, les mécanismes grammaticaux sont utilisés pour définir toutes les actions d'exécution des différentes instructions du langage ; en particulier une arithmétique entière est établie.

Dans un tel cas, l'ensemble des notions syntaxiques, l'ensemble des valeurs sémantiques et la fonction de correspondance sont définis par le même mécanisme grammatical.

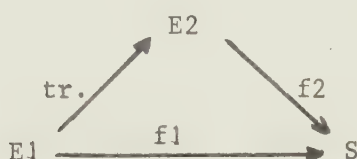
Mais on peut douter de la généralisation à la définition de langages plus puissants que le langage décrit en [55], qui est particulièrement simple.

## 6. METHODES ALGORITHMIQUES DE DEFINITION TRANDUCTIVE :

## METHODE DE WIRTH-WEBER ET METHODE DES GRAMMAIRES AFFIXES.

6.1. Introduction.

Ce chapitre est consacré aux méthodes traductives de définition de langages. Ces méthodes font porter l'essentiel de leurs efforts sur la définition de l'ensemble des règles d'écritures et sur la fonction  $f$  de correspondance. Il s'agit, ici, d'une fonction de traduction de programmes du langage en des programmes d'un autre langage que l'on peut définir à son tour. On peut schématiser ceci par :



Les informaticiens connaissent bien l'utilisation que l'on peut faire de langages intermédiaires qui peuvent être les mêmes pour des langages de départ différents et simplifier d'autant le traitement global.

Nous allons étudier deux méthodes qui sont caractéristiques du traitement de la sémantique statique dirigé par la syntaxe. La méthode de Wirth et Weber [24] a été utilisée pour la définition de EULER dirigée par une analyse syntaxique ascendante. La méthode des grammaires affixes de Koster [5] [25] est utilisée dans le compilateur de compilateur CDL. Ces méthodes se retrouvent dans tous les compilateurs de compilateurs. Nous les avons déjà introduites au chapitre 2. Leur principe est d'enrichir la description syntaxique (en BNF par exemple) par des fonctions ou actions sémantiques qui sont exécutées au cours de l'analyse syntaxique. L'évaluation de la sémantique statique se fait en alternance avec l'analyse et ces méthodes sont donc algorithmiques et impératives.

6.2. Méthode de Wirth et Weber [9]6.2.1. *Principe.*

On définit l'environnement comme étant l'ensemble des informations sémantiques, accessibles par toute règle sémantique, et accumulées jusqu'à un certain point de l'analyse. Au début de l'évaluation, l'environnement est initialisé et va être modifié lors de chaque reconnaissance syntaxique. L'environnement final, atteint lors de la reconnaissance de l'axiome, définit la sémantique du programme.

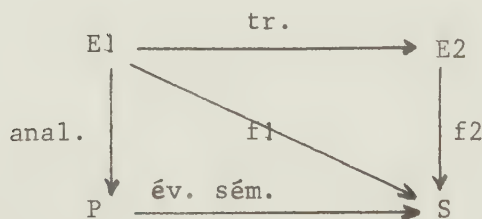
Cette méthode est, dans son principe, comparable à VDL à la différence fondamentale qu'elle définit la sémantique statique seulement, la sémantique dynamique pouvant être définie par un interprète comme celui du paragraphe 2.5.

On peut remarquer que, si dans la méthode de Wirth-Weber, l'évaluation sémantique est imbriquée à l'analyse syntaxique, plus généralement elle peut en être complètement dissociée :

Si on considère que l'analyse est une traduction du langage en une représentation post-fixée [26], on peut effectuer l'évaluation sémantique sur cette représentation, qui peut être considérée comme un langage intermédiaire pivot de la traduction. On a, dans ce cas :



Notre schéma du paragraphe précédent peut se transformer en :



Cette dissociation a l'avantage de rendre l'évaluation indépendante de l'analyse et donc de laisser le libre choix de la méthode d'analyse la mieux adaptée à la grammaire donnée.

### 6.2.2. Définition de TURINGOL.

L'environnement est composé de :

- V pile parallèle à la pile d'analyse, contient le texte symbolique des identificateurs reconnus et des informations de travail pour l'instruction if
- i index de V (géré par l'analyseur syntaxique)
- P programme généré
- k index de P
- N liste des identificateurs et informations associées.
- n index de N
- nt index de N, désigne le dernier symbole de ruban
- t index temporaire

Initialisations  $E_0 = \{V, O, P, \Omega, N, \Omega, \Omega, \Omega\}$

Le langage de définition utilisé est voisin de EULER

La définition sémantique identique est notée I .

```

<programme> ::= <ld><li>
                                     k ← k + 1
                                     P [k] ← 'stop'
/* recherche des instruction goto */
                                     t ← 1

L1 : if t = k then goto L3 ;
      if P [t][1] = 'goto' then goto L2 ;
      t ← t + 1
      goto L1 ;

L2 : if N [P [t] [2]] = 'ref'
      then error ('étiquette
                  indéfinie') ;

/* remplacement numéro étiquette par
   instruction */
P [t] [2] ← N [P [T] [3]] ;
      t ← t + 1 ;
      goto L1 ;

L3 : /* fin */

```

<ld> ::= <d>

```

    nt ← t ;
/* création de nt, limite des
   symboles de ruban */

```

<d> ::= TAPE ALPHABET IS id

```

    n ← 1 ;
/* première déclaration */
    N [1] ← V[i]

```

<d> ::= <d>, id

```

    t ← n
/* recherche de double définition */
L1 : if t = 0 then goto L2 ;
    if N [t] [1] = V [i] then
        error('double déclaration') ;
        t ← t-1
    goto L1 ;
L2 : n ← n + 1
    N [n] ← V [i]

```

<et> ::= id

```

    t ← n
/* recherche de double définition
   d'étiquette ou correction d'une
   référence anticipée */.

```

```

L1 : if t = nt then goto L3 ;
    if N [t] [1] = V [i] then goto L2 ;
    t ← t-1 ;
    goto L1 ;

```

```

L2 : if N [t] [2] = 'label' then
    error('double définition') ;
/* donc référence anticipée */
    N [t] [2] ← 'label' ;
    N [t] [3] ← k + 1 ;
    goto L4 ;

```

```

L3 : /* étiquette définie */
    n ← n + 1 ;
    N [n] ← (V[i] , 'label', k + 1) ;

```

```

                                L4 :
<let> ::= <et> :                I
<let> ::= <let> <et> :          I
<i>    ::= <let> <ins>          I
<i>    ::= <ins>                I

<ins>  ::= GOTO id              t ← n
                                /* recherche de l'étiquette et mise en
                                place comme référence anticipée
                                éventuellement */

L1 : if t = nt    then goto L2 ;
      if N [t] [1] = V [i] then goto L3 ;
      t ← t-1 ;
      goto L1 ;

L2 :   n ← n + 1 ;
      N [n] ← (V [i] , 'ref', Ω ) ;
      t ← n ;

L3 :   k ← k + 1 ;
      P [k] = ('jump', t) ;

<symb> ::= id                  t ← nt ;
                                /* recherche si symbole déclaré */
L1 : if t = 0 then error ('symbole indéfini') ;
      if N [t] = V [i] then goto L2 ;
      t ← t - 1 ;
      goto L1 ;

L2 :

<ins>  ::= PRINT <symb>        k ← k + 1 ;
                                P [k] ← ('print', k) ;

```

```

<ift> ::= IF THE TAPE SYMBOL      <symbol> THEN
                                     k ← k + 1 ;
                                     P [k] ← ('if', t, Ω) ;
                                     V [i] ← k ;
                                     /* conservation du rang de l'instruction
                                     if pour correction ultérieure */
<ins>  ::= <ift> <i>
                                     P [V [i-1]] [3] ← k + 1 ;
                                     /* correction */

<ins>  ::=
                                     I

<ins>  ::= (<li>)
                                     I

<ins>  ::= MOVE <sens> ONE SQUARE   k ← k + 1 ;
                                     P [k] ← ('move', s) ;

<sens> ::= LEFT
                                     s ← 1

<sens> ::= RIGHT
                                     s ← -1

```

### 6.2.3. Critiques de la méthode.

Cette méthode ne satisfait ni les théoriciens ni les usagers. Les théoriciens lui reprochent de ne pas donner une expression de la sémantique d'un programme mais seulement une représentation sous une autre forme. Nous avons déjà abordé ce point au paragraphe 4.4. et nous n'y reviendrons pas.

Les usagers, s'ils sont les premiers intéressés à une solution traductive pour des raisons économiques, ne trouvent absolument pas dans une telle définition une quelconque référence sur la signification de leurs programmes.

Voyons maintenant l'opinion des planteurs. On peut s'étonner de la présence de certaines productions (par exemple *ift*) dont l'intérêt n'est pas évident. Ces productions n'ont de raison d'être que parce que la méthode est impérative. En règle générale, si dans une production, plusieurs informations sémantiques apparaissent et doivent être exprimées à l'intérieur de la production, il est nécessaire de casser la production en plusieurs productions pour le faire. On voit alors apparaître un certain nombre de symboles non terminaux parasites introduits pour des raisons purement sémantiques et ce point nous semble très gênant car la description syntaxique doit être modelée

pour que l'algorithme d'évaluation sémantique puisse se dérouler correctement.

Une autre critique de cette méthode est l'utilisation d'un environnement global, qui est modifié à chaque pas de reconnaissance. Comment peut-on avoir l'assurance que l'environnement acquis à un point de l'analyse et qui sert de base à la définition sémantique, est cohérent, c'est-à-dire contient effectivement les informations sémantiques voulues ?

Un exemple de cette difficulté de garantir la cohérence est l'instruction `if` où l'on doit mémoriser une information à modifier par la suite.

La définition syntaxique du `if` est récursive et il est nécessaire de prévoir une structure d'environnement imbriquée pour en tenir compte. La pile `V`, parallèle à la pile syntaxique est utilisée à cet effet.

Mais peut-on avoir l'assurance qu'aucune autre structure syntaxique récursive ayant des propriétés sémantiques imbriquées n'a été oubliée ?

Un autre exemple, en apparence plus simple, est l'utilisation d'une variable de l'environnement qui est supposée représenter une certaine information sémantique. La représente-t-elle vraiment ? dans tous les cas ? N'a-t-elle pas été utilisée à d'autres fins depuis sa définition ?

Ces questions que connaissent bien les "compileurs",\* sont sans réponse. Des tests approfondis sont les seules possibilités de vérification. Une solide expérience peut apporter une certaine foi en une définition mais il est certain que cette méthode est insuffisante pour obtenir la garantie que l'implantation décrite est conforme à ce que l'on souhaitait.

Une autre critique, illustrée par le traitement des étiquettes dans TURINGOL ou par le traitement de l'instruction `"if"` est la limitation à des traitements à une passe. L'élaboration de l'environnement se fait dans un ordre strictement gauche droite et le traitement des références anticipées\*\* de toute nature (par exemple la référence à l'instruction qui suit le corps d'un `"if"`) est impossible, sinon par des artifices souvent lourds. Or cette notion est présente dans tous les langages, même les plus simples ou les mieux structurés, elle est inhérente au fait qu'on ne peut se limiter à un flot de contrôle purement séquentiel. La présence d'instructions de branchement n'est ici qu'un élément perturbateur supplémentaire.

---

\* qu'on nous excuse ce néologisme-barbarisme, mais comment désigner simplement les "personnes qui écrivent des compilateurs" ?

\*\* nous entendons par référence anticipée la référence à une information non encore connue.

Toutes ces critiques ne suffisent pas à annuler l'intérêt pratique de cette méthode qui est, actuellement, la base d'écriture de beaucoup de compilateurs : pour la raison bien simple et évidente que la définition du compilateur se construit avec la définition sémantique du langage, qu'elle en est même la raison essentielle.

Mais les compilateurs auraient tort de penser que la bonne connaissance et la pratique quotidienne de ces méthodes soient un aboutissement pour eux. La manipulation d'un environnement global dans un gros compilateur conduit bien vite à une écriture approximative et par nature peu sûre. L'objectif central de cette thèse est bien de les convaincre que des méthodes plus élaborées, apportant une sécurité sans commune mesure et par laquelle des preuves de correction sont possibles, ne sont pas d'un coût exorbitant et que la sécurité garantie pour les traducteurs est un facteur de progrès pour l'informatique quotidienne.

### 6.3. Méthode des grammaires affixes [5][25]

#### 6.3.1. *Introduction.*

L'idée de base des grammaires affixes s'exprime à travers le langage CDL conçu pour la définition de compilateurs. Elle met en évidence un point fondamental que nous avons déjà souligné dans la méthode de Wirth-Weber : la construction d'un traducteur autour de la définition syntaxique du langage à traduire. CDL est donc, au départ, un langage de description syntaxique. Mais la définition n'est plus dirigée par la syntaxe, mais dirigée par l'analyse syntaxique. La méthode d'analyse est, dans ce cas, exclusivement du type descendant déterministe, bien qu'il soit possible par des retours-arrière, de traiter des grammaires non déterministes. La description de l'analyseur, et donc de la syntaxe du langage, est étendue par l'incorporation d'actions et de prédicats affixés qui permettent d'exprimer un certain nombre de contraintes contextuelles et d'effectuer la traduction.

#### 6.3.2. *Principe.*

Les prédicats sont des procédures booléennes associées aux symboles terminaux et non terminaux de la grammaire. Avec la condition que la grammaire est LL(1) [27], ces prédicats assurent la reconnaissance du texte à traduire par une analyse descendante déterministe. Les affixes sont les paramètres (notés +) et les variables locales (notés -) associés aux prédicats. Ils permettent de communiquer des informations aux procédures booléennes que sont les prédicats (paramètres d'entrée d'héritage d'information)

et de retourner des informations au point d'appel (paramètres de sortie de synthèse d'information).

Les actions sont des prédicats spéciaux ne retournant pas de valeurs booléennes: elles permettent d'effectuer ponctuellement un certain nombre de traitements paramétrisés en entrée et en sortie.

Ces définitions constituent la base des grammaires affixes et du langage CDL qui est leur langage de formulation.

### 6.3.3. Exemples.

- i) Le premier exemple d'illustration des grammaires affixes est l'exemple 2 du chapitre 5 : les suites de deux lettres dont la première précède alphabétiquement la seconde.

*macro pointer* LA = 10 , LZ = 35 .

cette instruction définit LA et LZ comme étant les codes des lettres.

*external action* PASLET,PASINF .

On définit ces actions externes qui

- imprime le message : "le caractère n'est pas une lettre"

- imprime le message : "ordre alphabétique incorrect"

*macro predicate* ISINF = '1' < '2' .

*macro predicate* ISLET = '1' ≥ LA & '1' ≤ LZ .

On définit deux prédicats de tests ('i' désigne le i<sup>e</sup> paramètre du prédicat).

Dans les prédicats suivants, la méta-syntaxe utilisée est celle de CDL qui est inspirée de la méta-syntaxe des W grammaires.

Ce n'est pas un hasard comme nous le verrons au chapitre 8.

CHAINE + L + M : (ISLET + L, TEST + L + M ; PASLET + L).

TEST + L + M : (ISLET + M, (ISINF + L + M ; PASINF + L + M) ;  
PASLET + M) .

Ces prédicats peuvent se traduire en ALGOL 60, par exemple, par :

```
boolean procedure ISINF (L, M) ;
```

```
    ISINF := L < M
```

```
fin ISINF ;
```

```
boolean procedure ISLET (L) ;
```

```
    ISLET := L ≥ 10 & L ≤ 35
```

```
fin ISLET ;
```

```

booleen procedure CHAINE (L, M) ;
    CHAINE := si ISLET (L) alors TEST (L, M)
                sinon PASLET (L)

    fin CHAINE ;

booleen procedure TEST (L, M) ;
    TEST := si ISLET (M) alors
        debut
            si ISINF (L, M) alors vrai
                sinon PASINF (L, M)
            fin
        sinon PASLET (M)

    fin TEST ;

```

Suivons le traitement des chaînes b1 puis 1b (code de b = 11, code de 1 = 21)

. CHAINE + b + 1 : (ISLET + b, TEST + b + 1 ; PASLET + b) .

- le prédicat ISLET + b donne  $ISLET = 11 \geq 10 \ \& \ 11 \leq 35$  . = vrai

- le prédicat TEST + b + 1 donne :

TEST + b + 1 : (ISLET + 1, (ISINF + b + 1 ; PASINF + b + 1) ; PASLET + 1 .

- le prédicat ISLET + 1 donne  $ISLET = 21 \geq 10 \ \& \ 21 \leq 35$  . = vrai

- le prédicat ISINF + b + 1 donne  $ISINF = 11 < 21$  . = vrai

donc le prédicat CHAINE + b + 1 est vrai

. CHAINE + 1 + b : (ISLET + 1 , TEST + 1 + b ; PASLET + 1) .

- le prédicat ISLET + 1 donne  $ISLET = 21 \geq 10 \ \& \ 21 \leq 35$  . = vrai

- le prédicat TEST + 1 + b donne :

. TEST + 1 + b : (ISLET + b , (ISINF + 1 + b ; PASINF + 1 + b) ; PASLET + b .

. le prédicat ISLET + b donne  $ISLET = 11 \geq 10 \ \& \ 11 \leq 35$  . = vrai

- le prédicat ISINF + 1 + b donne  $ISINF = 21 < 11$  . = faux

- l'action PASINF + 1 + b imprime le message :

"ordre alphabétique incorrect entre 1 et b".

ii) Ce deuxième exemple illustre la déclaration des symboles de ruban de TURINGOL.

Les prédicats et affixes utilisés sont :

DECL + T            analyse et construit une table T de symboles.  
 DCL + D            analyse une déclaration, la transmet dans D.  
 TAPE                analyse TAPE ALPHABET IS  
 VIRG                analyse une virgule  
 SDCL + X + Y       analyse une suite de déclarations. Au début de l'analyse la  
                       table des symboles est X, à la fin Y.  
 TEST + D + X       teste si la déclaration D ne figure pas déjà dans X.

Grammaire affixe :

DECL + T - D : TAPE, DCL + D, SDCL + D + T .  
 SDCL + X + Y - D : (VIRG, DCL + D, TEST + D + X,  
                       FAIRE + Y + X + D, SDCL + Y + Y ;  
                       FIN + Y + X) .

Les actions et affixes sont :

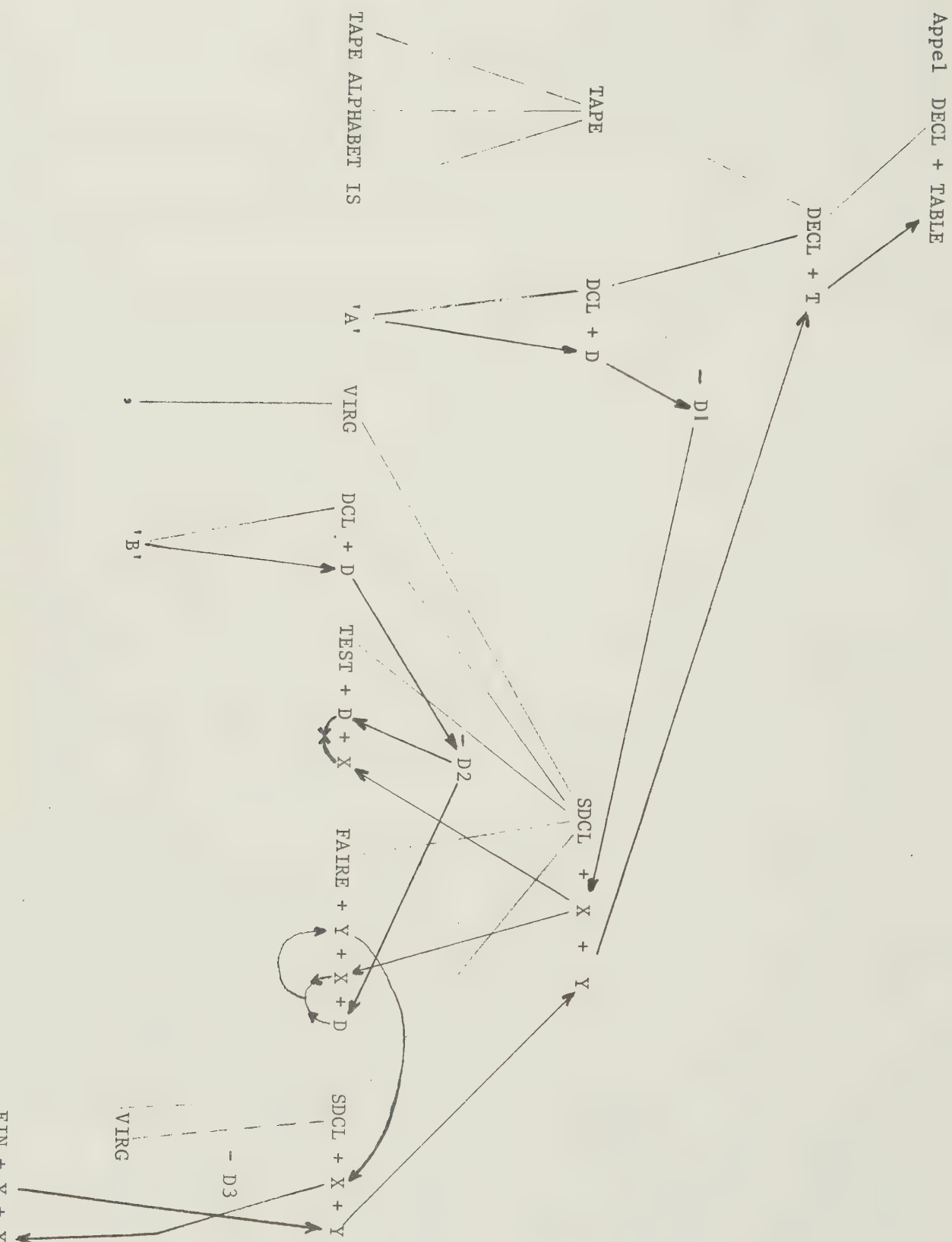
FAIRE + Y + X + D            Y := X ∪ D  
 FIN + Y + X                   Y := X

La figure ci-jointe décrit le flot d'informations engendré par la déclaration

TAPE ALPHABET IS A , B

#### 6.3.4. Critiques de la méthode.

La première critique est que cette méthode est dirigée par l'analyse syntaxique et que l'on s'interdit donc d'utiliser la méthode d'analyse la mieux adaptée à la grammaire donnée. Cette critique devient plus vive quand l'expérience montre que la méthode d'analyse utilisée impose bien souvent des contraintes de description difficilement supportables. Ensuite, on peut critiquer le mélange qui est fait entre prédicats d'analyse et prédicats ou actions sémantiques. Là encore, les praticiens divergent des théoriciens car ils estiment que c'est une facilité importante. On peut opposer ces inclusions, assez naturelles dans une méthode d'analyse descendante, avec les actions nécessairement externes à l'analyse des méthodes ascendantes. Il est peut être plus naturel d'insérer une action à l'endroit précis où elle doit être activée, plutôt que de créer des productions artificielles pour le faire.



Mais, dans tous les cas, des méthodes dirigées par l'analyse syntaxique sont criticables car trop particularisées.

La troisième critique est plus importante. Les affixes sont indifféremment paramètres d'entrée ou de sortie, et rien ne permet de distinguer les uns des autres. Une preuve en est dans les procédures ALGOL60 du paragraphe précédent : les affixes (+) sont transformés en paramètres de procédure, mais paramètres par nom, c'est-à-dire peuvent servir indifféremment et sans contrôle possible, à passer des informations du point d'appel vers la procédure et / ou de la procédure vers le point d'appel.

Une solution partielle, pour les affixes d'entrée seulement, est d'en faire des paramètres par valeur. Cette distinction est pourtant fondamentale car elle seule permettrait un contrôle complet du flot d'informations mis en oeuvre au cours de l'analyse. D'autre part, on ne peut avoir aucune garantie qu'un affixe d'entrée est connu au moment de l'appel d'un prédicat ou d'une action, pas plus que les valeurs de retour sont effectivement passées aux affixes de sorties.

Enfin, la méthode d'analyse gauche droite et le mode d'héritage d'information par paramètre d'entrée et de synthèse par paramètre de sortie limite naturellement le flot sémantique à être strictement gauche droite et permet difficilement le traitement de référence anticipées.

En conclusion, nous pouvons dire que cette méthode introduit la synthèse et l'héritage d'informations mais de manière peu sûre et difficilement contrôlable car trop implicite, et que le flot d'information est limité à un ordre gauche droite insuffisant même dans les langages les plus simples. Il est néanmoins certain que cette méthode est extrêmement précieuse pratiquement car elle permet de localiser les informations manipulées et que les structures syntaxiques récursives conduisent à des structures d'actions naturellement imbriquées par le jeu de procédures récursives. Elle est donc plus satisfaisante que la méthode de Wirth-Weber en limitant l'environnement global et en évitant de le gérer explicitement pour des structures imbriquées.

Le lecteur attentif aura remarqué les points communs avec l'approche grammaticale par doubles grammaires. La comparaison entre ces méthodes et la méthode des attributs sémantiques est faite au chapitre 8.

## 7. METHODE DECLARATIVE DE DEFINITION TRANDUCTIVE : METHODE DES ATTRIBUTS SEMANTIQUES [1] , [28]

### 7.1. Introduction.

Cette méthode, introduite par KNUTH , reprend la plupart des idées que nous avons découvertes au chapitre 6 en les explicitant et en les systématisant.

Elle est tout d'abord dirigée par la syntaxe, c'est-à-dire que la définition sémantique accompagne chaque règle de description syntaxique. Mais les dangers de l'environnement global sont supprimés par une stricte localisation des informations sémantiques. Chaque production a son environnement propre. En étant encore plus restrictif, c'est chaque symbole non terminal qui a un environnement. Nous avons aussi critiqué la juxtaposition, pèle mèle, d'un ensemble d'informations dans l'environnement : à chaque information est associé un attribut sémantique de définition.

Nous avons également critiqué le flot d'informations des grammaires affixes en ce qu'il a d'imprécis, car rien ne ressemble plus à un affixe d'entrée qu'un affixe de sortie : un attribut est hérité ou synthétisé exclusivement.

Enfin, nous avons regretté l'aspect impératif des deux méthodes précédentes : cette méthode est déclarative en ce sens qu'elle ne suppose rien sur la méthode d'analyse syntaxique et qu'elle n'impose, en aucun cas, un ordre de traitement gauche-droite. L'ordre de parcours des informations sémantiques peut être gauche-droite pour certains attributs et droite-gauche pour d'autres.

Ainsi, cette méthode traductive permet-elle de définir des processus de traduction à nombre quelconque de passes.

Ce chapitre est consacré à une présentation informelle de la méthode à travers deux exemples dont le langage TURINGOL.

### 7.2. Présentation informelle de la méthode.

Une présentation sémantique par attributs s'articule autour de la grammaire hors contexte définissant la syntaxe du langage.

Des règles de définition sont associées à chaque règle syntaxique pour chaque attribut. La méthode est de nature traductive, c'est-à-dire que

l'ensemble S est supposé connu, les différents concepts sémantiques étant définis dans chaque règle de correspondance. En particulier, la traduction d'un programme est un de ces concepts et est, naturellement, un des attributs synthétisés de l'axiome de la grammaire.

L'arbre syntaxique d'un programme est décoré par l'ensemble des informations sémantiques associées aux noeuds de l'arbre. Le but de l'évaluation de toutes ces informations est d'obtenir la valeur sémantique du programme qui est l'attribut traduction de la racine de l'arbre syntaxique.

Donnons deux exemples qui présentent la méthode de façon intuitive.

*Exemple 1* : Algorithme de Sethi-Ullman [29]

Cet algorithme permet de déterminer le nombre de registres nécessaires à l'évaluation d'une expression arithmétique, si on se limite à des calculateurs ayant des instructions du type :

1<sup>r</sup> opérande en registre, 2<sup>e</sup> en mémoire ou en registre.

Nous nous limiterons aux opérateurs d'addition et de soustraction (l'extension à d'autres opérateurs est immédiate ; on ne tient pas compte, ici, de la commutativité de l'addition).

La nature des instructions machine nous impose de connaître si un opérande est gauche (il doit être dans un registre) ou droit (il peut être en mémoire). Cette information est transmise par l'intermédiaire d'un attribut hérité noté g, booléen, vrai si l'opérande est gauche, faux sinon. Le nombre de registres est transmis par l'attribut synthétisé r.

r(programme) constitue le but de la définition sémantique.

|                                               |                                      |
|-----------------------------------------------|--------------------------------------|
| <programme> ::= <expression>                  | r(programme) := r(expression)        |
|                                               | g(expression) := vrai                |
| <expression> ::= <opérande>                   | r(expression) := r(opérande)         |
|                                               | g(opérande) := g(expression)         |
| <expression> ::= <expression> <pm> <opérande> | g(expression') := vrai               |
|                                               | g(opérande) := faux                  |
|                                               | r(expression) := si r(expression') = |
|                                               | r(opérande) alors                    |
|                                               | r(expression') + 1                   |
|                                               | sinon max(r(expression'),            |
|                                               | r(opérande))                         |

```
<opérande> ::= id          r(opérande) := si g(opérande) alors 1
                                     sinon 0
```

[illegible]
$$\langle pm \rangle := + \mid -$$

*Note* : Lorsqu'un même symbole apparaît plusieurs fois dans une règle, la désignation de chaque occurrence est faite, de gauche à droite dans la règle, par le nom du symbole pour l'occurrence la plus à gauche, puis par le nom primé, etc.

### Exemple 2 : TURINGOL

La définition de notre langage préféré a pour but de définir la traduction de programmes TURINGOL en programme TL/1 correspondants.

Les attributs utilisés sont :

|       |            |                                                                                                              |
|-------|------------|--------------------------------------------------------------------------------------------------------------|
| p     | hérité     | porte le numéro de la lère instruction TL/1 correspondant à une instruction TURINGOL                         |
| s     | synthétisé | porte le numéro de l'instruction TL/1 suivant les instructions TL/1 correspondant à une instruction TURINGOL |
| et    | hérité     | étiquettes déjà déclarées, associées à un numéro d'instruction TL/1                                          |
| étiq  | synthétisé | id                                                                                                           |
| tape  | synthétisé | symboles de ruban déjà déclarés, associés à un numéro d'ordre                                                |
| n     | synthétisé | numéro d'ordre d'un symbole de ruban                                                                         |
| gtape | hérité     | ensemble des symboles de ruban                                                                               |
| getiq | hérité     | ensemble des étiquettes                                                                                      |
| tl    | synthétisé | ensemble des instructions TL/1 correspondant à une instruction TURINGOL                                      |
| gd    | synthétisé | donne le sens d'un MOVE                                                                                      |
| text  | synthétisé | donne la chaîne de caractères constituant le symbole terminal id.                                            |

La traduction d'un programme est donnée par  $t_1(\text{programme})$

```

<programme> ::= <déclaration> ; <liste> .
               gtape(liste) := gtape(déclaration)
               getiq(liste) := etiq(liste)
               tl(programme) := tl(liste) u (s(liste) , stop)
               p(liste) := 1
               s(programme) := s(liste)
               et(liste) :=  $\phi$ 

<déclaration> ::= TAPE ALPHABET IS id
               n(déclaration) := 1
               tape(déclaration) := (text(id), n (déclaration))

<déclaration> ::= <déclaration> , id
               si text(id)  $\in$  tape(déclaration') alors
                   erreur('double déclaration')
               n(déclaration) := n(déclaration') + 1
               tape(déclaration) := tape(déclaration') u (text(id) ,
                                                           n(déclaration))

<liste> ::= <instruction>
               p(instruction) := p(liste)
               s(liste) := s(instruction)
               et(instruction) := et(liste)
               etiq(liste) := etiq(instruction)
               gtape(instruction) := gtape(liste)
               getiq(instruction) := getiq(liste)
               tl(liste) := tl(instruction)

<liste> ::= <liste> ; <instruction>
               p(liste) := p(liste)
               p(instruction) := s(liste')
               s(liste) := s(instruction)
               et(liste') := et(liste)
               et(instruction) := etiq(liste')
               etiq(liste) := etiq(instruction)
               gtape(liste') := gtape(liste)
               gtape(instruction) := gtape(liste)

```

```

et(instruction) := et(ins)
etiq(ins) := etiq(instruction)
getiq(instruction) := getiq(ins)
gtape(instruction) := gtape(ins)

<ins> ::=

s(ins) := p (ins)
etiq(ins) := et(ins)
tl(ins) :=  $\phi$ 

<ins> ::= MOVE <sens> ONE SQUARE

tl(ins) := (p(ins), move, gd(sens))
s(ins) := p(ins) + 1
etiq(ins) := et(ins)

<sens> ::= LEFT

gd(sens) := 1

<sens> ::= RIGHT

gd(sens) := -1

<ins> ::= GOTO id

si text(id)  $\neq$  getiq(ins)
alors erreur ('étiquette indéfinie')
tl(ins) := (p(ins), jump, getiq(ins)(text(id), ?))
s(ins) := p(ins) + 1
etiq(ins) := et(ins)

<ins> ::= ( <liste> )

p(liste) := p(ins)
s(ins) := s(liste)
et(liste) := et(ins)
etiq(ins) := etiq(liste)
tl(ins) := tl(liste)
getiq(liste) := getiq(ins)
gtape(liste) := gtape(ins)

```

```

getiq(liste') := getiq(liste)
getiq(instruction) := getiq(liste)
tl(liste) := tl(liste') ∪ tl(instruction)

<instruction> ::= id = <instruction>
    p(instruction') := p(instruction)
    s(instruction) := s(instruction')
    si text(id) ≠ et(instruction) alors
        erreur('double définition d'étiquette')
    et(instruction') := et(instruction ∪ (text(id), p(instruction)))
    etiq(instruction) := etiq(instruction')
    gtape(instruction') := gtape(instruction)
    getiq(instruction') := getiq(instruction)
    tl(instruction) := tl(instruction')

<instruction> ::= <ins>
    p(ins) := p(instruction)
    s(instruction) := s(ins)
    et(ins) := et(instruction)
    etiq(instruction) := etiq(ins)
    gtape(ins) := gtape(instruction)
    getiq(ins) := getiq(instruction)
    tl(instruction) := tl(ins)

<ins> ::= PRINT id
    si text(id) ≠ gtape(ins) alors
        erreur('symbole indéfini')
    tl(ins) := (p(ins), print, gtape(ins) (text(id))
    s(ins) := p(ins) + 1
    etiq(ins) := et(ins)

<ins> ::= IF THE TAPE SYMBOL IS id THEN <instruction>
    si text(id) ≠ gtape(ins) alors
        erreur('symbole indéfini')
    tl(ins) := (p(ins), if, gtape(ins)(text(id) , ?),
        s(instruction)) ∪ tl(instruction)
    p(instruction := p(ins) + 1
    s(ins) := s(instruction)

```

*Note :* ? signale le champ recherché dans les attributs structurés  
gtape ou getiq.

### 7.3. Intérêt de la méthode.

Les critiques des théoriciens et des usagers pour les méthodes traductives ont été faites au chapitre précédent. Elles demeurent sur le fait qu'il ne s'agit que d'une traduction mais nous avons défendu ce point de vue au § 1.4., un langage de sémantique supposée connue, ou dont on peut aussi définir la sémantique, étant une base de définition acceptable et offrant l'énorme avantage de tenir compte des réalités informatiques.

Cette méthode présente indiscutablement des avantages sur les autres méthodes traductives. La définition sémantique est claire, lisible, naturelle et précise. Elle est de plus sans limite, c'est-à-dire qu'il est possible de créer autant d'attributs que nécessaire pour exprimer les concepts sémantiques inhérents à un langage. La localisation et l'individualisation des définitions apportent, en outre, la possibilité d'exprimer clairement et simplement l'interdépendance entre tous ces concepts. De plus une telle définition est extrêmement modulaire : l'ajout d'un concept se fait sans remettre complètement en cause la définition. Par exemple, dans la définition sémantique de SIMULA67 de Wilner [30], l'instruction POUR est ajoutée dans une annexe sans modification du reste de la définition.

Nous avons déjà souligné que cette méthode est complètement indépendante de toute méthode d'analyse syntaxique. Dans le système FOLDS de Fang [31], des grammaires ambiguës sont prises en compte et les ambiguïtés sont levées sémantiquement par des attributs spéciaux. Dans le système SOFT du MBLE [32] la méthode est descendante déterministe, dans notre système, ascendante déterministe.

Un autre aspect fondamental est le caractère déclaratif de ces définitions. Nous avons signalé ce point sans peut-être l'éclairer complètement. La définition de TURINGOL nous permet maintenant de le faire : supposons qu'un esprit malicieux décide de modifier la définition de l'axiome de la façon suivante :

```
<programme> ::= <liste> ; <déclaration> .
```

c'est-à-dire que, maintenant, les déclarations des symboles de ruban sont en fin de programme. On peut imaginer les perturbations sans fin que cette modification imposerait à des méthodes algorithmiques, allant jusqu'à l'impossibilité complète de traitement pour une définition à une passe gauche-droite.

Avec la méthode des attributs, cette modification purement syntaxique ne modifie en rien la définition sémantique car les concepts sémantiques exprimés restent invariants. Il est certain que l'évaluation sémantique, c'est-à-dire la décoration de l'arbre de dérivation est profondément modifiée mais pas la définition.

Enfin, cette méthode apporte la définition d'un traducteur au sein de la définition sémantique d'un langage, ce qui résout le problème de l'obtention automatique d'une implantation.

#### 7.4. Critiques de la méthode.

La première critique [17] est que le caractère déclaratif de la méthode est atténué par la nécessité de penser à un ordre d'évaluation des attributs lors de la définition. Ce point de vue est vrai dans la mesure où la définition sémantique est associée à une règle de production décrivant la syntaxe concrète du langage. Mais il est également vrai que c'est le caractère synthétique d'une règle de production qui doit naturellement guider la définition sémantique. C'est la localisation des définitions d'attributs qui est critiquée car il est bien évident que l'emploi de productions récursives à droite et non plus à gauche, inverse naturellement l'ordre d'apparition des entités du langage et impose une redéfinition sémantique, car si une récursivité gauche s'arrête sur le premier symbole d'une liste, une récursivité droite s'arrête sur le dernier. Le fond de cette critique est l'emploi d'une syntaxe concrète et non abstraite. C'est un choix criticable mais simplificateur : il évite de donner la définition d'une transformation qui apporterait uniquement une systématisation accrue de la définition. Cette modification est prévue dans le système DELTA mais ne nous semble pas fondamentale.

Une critique plus forte est le nombre élevé d'informations sémantiques mises en oeuvre lors d'une évaluation sémantique. Le traitement d'un tel nombre d'attributs met en cause complètement l'applicabilité de la méthode malgré tous les avantages qu'elle apporte. Le chapitre 11 présente les méthodes qui permettent de réduire ce nombre en conservant toute la puissance et les avantages des attributs. Cette réduction dépasse 90% en moyenne et son coût ne remet pas en cause l'intérêt pratique de la méthode. Le chapitre 12 ouvre d'ailleurs des perspectives de compilation parallèle qui permettent à cette méthode d'être compétitive et même plus performante que les méthodes classiques de traduction du chapitre 6.

## 8. COMPARAISON ENTRE ATTRIBUTS SEMANTIQUES, DOUBLES GRAMMAIRES ET GRAMMAIRES AFFIXES

### 8.1. Introduction

Nous avons déjà souligné certains points communs et certaines divergences entre ces trois méthodes. Nous allons en donner, tout d'abord, une comparaison informelle, en choisissant un certain nombre de critères.

Ces trois méthodes ont, en commun, de permettre de définir et de manipuler des informations contextuelles. Le premier critère de comparaison est le type de définition, explicite ou implicite ; le second concerne le flot d'informations contextuelles qui peut être explicite, c'est-à-dire algorithmique, ou implicite ; ce flot permet également de synthétiser et/ou d'hériter des informations, c'est notre troisième critère ; enfin notre quatrième critère est l'ordre gauche droite, ou non, de ce flot.

Pour affiner cette comparaison, nous étudierons ensuite le passage éventuel de l'une à l'autre de ces méthodes, qui peut nous éclairer sur leur puissance descriptive.

### 8.2. Comparaison informelle

#### 8.2.1. *Type de définition.*

- . les définitions d'attributs sont explicites et faites en liaison avec une règle de syntaxe.
- . les métanotions des W-grammaires constituent une enveloppe des contraintes contextuelles : elles sont définies implicitement.
- . les affixes sont des paramètres de fonctions, définis à chaque occurrence d'appel de ces fonctions ; à ce titre, on peut considérer que ces paramètres définissent implicitement un relais vers les arguments d'appel, qui, par le biais des appels imbriqués, peuvent renvoyer, de proche en proche, vers la définition explicite des arguments.

#### 8.2.2. *Flot d'informations contextuelles.*

- . les attributs définissent implicitement le flot par leurs définitions.
- . les métanotions définissent implicitement le flot par la règle du remplacement uniforme.

- . les affixes sont paramètres d'entrée ou de sortie : ils définissent donc explicitement le flot d'informations.

#### 8.2.3. *Synthèse et héritage d'informations.*

- . les attributs autorisent la synthèse et l'héritage complets d'informations.
- . les notions de synthèse et d'héritage d'informations contextuelles sont absentes des W-grammaires. Cependant, les métanotations apparaissant dans une hyper-règle correspondent au cours d'une évaluation effective d'un programme, à l'idée de synthèse et d'héritage. La règle du remplacement uniforme assure la transmission d'informations pour les différentes occurrences de la même métanotation.
- . les affixes d'entrée servent à l'héritage, ceux de sortie à la synthèse d'informations.

#### 8.2.4. *Ordre du flot.*

- . par attributs, le flot est non limité ; il peut se faire en tous sens.
- . pour les doubles grammaires, l'ordre du flot est quelconque : dans une hyper-règle, de toutes les occurrences d'une métanotation, on ne sait pas laquelle effectue la synthèse et déclenche l'héritage, si même il y en a une.
- . pour les grammaires affixes, la nature algorithmique de la méthode impose un flot strictement gauche droite.

### 8.3 Comparaison des méthodes deux à deux

Nous nous intéressons, ici, à la comparaison de puissance des méthodes, en analysant sous quelles conditions on peut automatiser le passage de l'une à l'autre.

#### 8.3.1. *Attributs sémantiques et grammaires affixes.*

Nous venons de voir les différences essentielles entre les deux méthodes.

Le passage automatique d'une définition par attributs à une grammaire affixe n'est pas possible dans le cas le plus général. Si, naturellement, on peut associer un attribut hérité à un affixe d'entrée et un attribut synthétisé à un affixe de sortie, le flot de ces informations est quelconque, par

attributs, et strictement gauche droite, par grammaire affixe.

Cependant, nous verrons (§11.4.4) que l'on peut apporter des restrictions à une définition par attributs pour pouvoir effectuer l'évaluation sémantique au cours de l'analyse syntaxique descendante gauche-droite. Ces restrictions limitent le flot sémantique à être strictement gauche-droite.

Un autre problème doit être résolu pour automatiser le passage, celui de la définition syntaxique du langage qui doit, impérativement, être donnée par une grammaire LL(1) pour les grammaires affixes. Cette restriction vient de la nature algorithmique de la méthode et peut être vérifiée sans difficulté [27].

Illustrons la transformation à l'aide de l'algorithme de Sethi-Ullman du §7.2. dont nous avons modifié la grammaire pour la rendre LL(1).

PROGRAMME + R :

EXPRESSION + vrai + R.

EXPRESSION + G + R - R1 - R2 :

OPERANDE + vrai + R1,

PLUS, EXPRESSION + faux + R2,

MAX+R+R1+R2 ;

OPERANDE + G + R.

OPERANDE + G + R :

ID, R1 OU 0 + G + R ;

PARG, EXPRESSION + G + R, PARD.

MAX+R1+R2+R3 :

R1 := si R2 = R3 alors R2+1

sinon max (R2,R3).

R1 OU 0 + G + R :

R := si G alors 1 sinon 0.

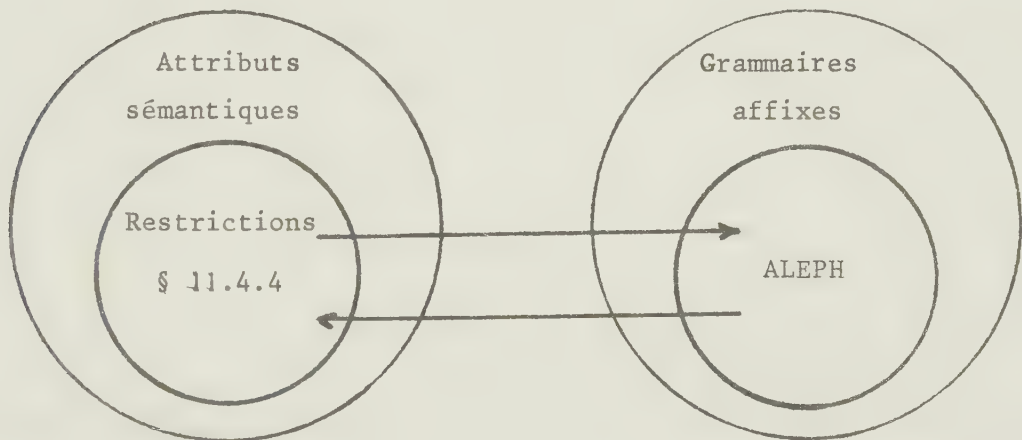
Les prédicats d'analyse PLUS, PARG, PARD, ID ne sont pas définis ici.

Voyons maintenant le passage inverse grammaire affixe → attributs. Le problème du flot n'existe pas dans ce cas. Seul subsiste la conversion des affiches en attributs. Nous avons déjà souligné l'impossibilité, dans le cas général, de déterminer si un affixe synthétise ou hérite de l'information, d'autant plus qu'il peut réaliser alternativement les opérations de synthèse et d'héritage.

On doit noter que cette indétermination sur le sens de parcours des informations pour les grammaires affixes est partiellement levée dans une version plus récente du langage CDL, le langage ALEPH, où il est possible de préciser la nature synthétisée ou héritée d'un affixe.

Si on impose cette restriction à une définition par grammaire affixe, le problème de la conversion affixe-attribut est levé et le passage grammaire affixe  $\rightarrow$  attributs est immédiat.

On peut établir le schéma de correspondance suivant entre les deux méthodes :



### 8.3.2. *W-grammaires et attributs sémantiques.*

Les points communs entre les deux méthodes sont nombreux, notamment la nature déclarative et locale des deux définitions. Un exemple pour les *W-grammaires* est la définition de Turingol : laissons le même esprit malicieux que précédemment modifier le langage pour placer la déclaration du vocabulaire de ruban en fin de programme ; la seule modification à apporter à la définition est l'hyper-règle programme qui devient :

```
programme : TAPE <LABETY> <LABETY> instruction suite,
           ou LABETY unique, declare symbol,
           declaration de TAPE.
```

La différence fondamentale entre les deux méthodes est mise en évidence par cette hyper-règle :

- . des deux métanotions TAPE, laquelle synthétise l'information pour l'hériter dans l'autre ?
- . des trois méta-notions LABETY, celle apparaissant dans un prédicat ne peut être qu'héritée. Mais pour les deux autres, la même question peut être posée.

C'est ce manque d'information sur la nature synthétisée ou héritée des méta-notions qui rend une définition par W-grammaires si difficile à suivre surtout et qui limite l'implantation automatique à partir des définitions. Mais c'est certainement aussi cette particularité qui donne aux W-grammaires leur puissance descriptive.

On peut se demander s'il n'est pas possible de passer d'une définition par attributs sémantiques à une définition par doubles grammaires et inversement.

Le passage attributs → doubles grammaires ne pose que peu de problèmes si on limite les attributs à être des chaînes de caractères.

L'exemple de l'algorithme de Sethi-Ullman (§7.2.) nous servira de support à cette transformation.

Les concepts sémantiques manipulés sont les attributs r et g. Nous décrivons leurs ensembles de valeurs par des métanotions :

VF :: vrai ; faux.

N :: un ; N un.

NETY :: N ; EMPTY.

EMPTY :: .

Un problème se pose pour représenter l'arithmétique entière nécessaire pour le calcul du nombre de registres : ce compte se fait par le nombre de fois où la notion "un" apparaît dans une dérivation terminale de N.

La grammaire hors contexte utilisée pour les attributs est directement reprise pour les doubles grammaires, ses attributs accompagnant chaque symbole pour former une hyper-notion.

La double grammaire obtenue à partir de la définition par attributs du §7.2. est :

```

programme avec  $r = N$  : expression      avec  $g = \text{vrai}$  et  $r = N$ .
expression avec  $g = \text{VF}$  et  $r = N1$  : op  rande avec  $g = \text{VF}$  et  $r = N1$  ;
                                expression avec  $g = \text{vrai}$  et  $r = N2$ ,
                                pm,
                                op  rande avec  $g = \text{faux}$  et  $r = \text{NETY}$ ,
                                ou  $N1 = \max(N2, \text{NETY})$ .
op  rande avec  $g = \text{VF}$  et  $r = \text{NETY}$  : parg symbol,
                                expression avec  $g = \text{VF}$  et  $r = \text{NETY}$ ,
                                pard symbol ;
                                id symbol,
                                ou feuille VF et NETY.

ou feuille vrai et un : EMPTY.
ou feuille faux et EMPTY : EMPTY.
pm : plus symbol ; moins symbol.
ou  $N1 = \max(N2, \text{NETY})$  : ou  $N1 = N2$ , ou  $N2 \supset \text{NETY}$  ;
                                ou  $N1 = \text{NETY}$ , ou  $\text{NETY} \supset N2$  ;
                                ou  $N1 = N2$  un.

ou  $N1 \ N2 \supset N1$  : EMPTY.
ou  $N1 \supset \text{EMPTY}$  : EMPTY.
ou  $N = N$  : EMPTY.

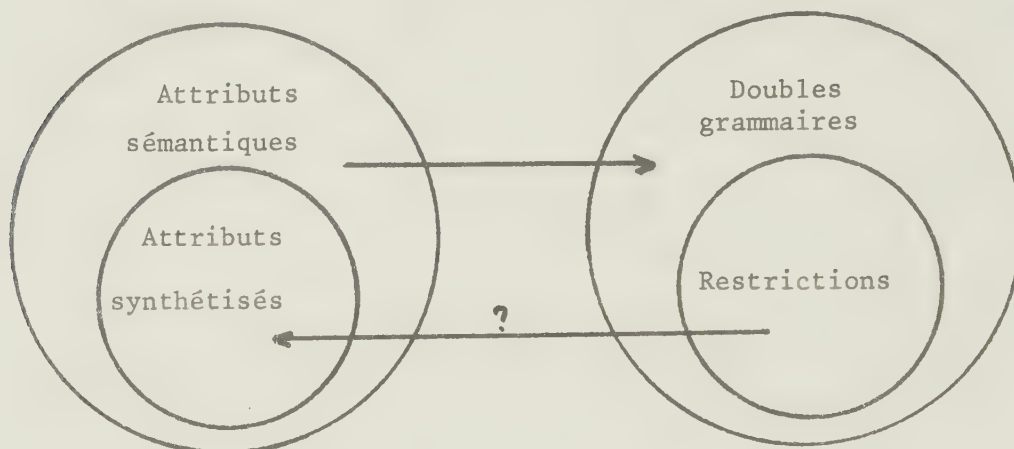
```

Dans cette définition, nous avons pris quelques libertés syntaxiques avec la méthode des doubles grammaires pour calquer au maximum la définition sur celle par attributs sémantiques.

A la définition des prédicats près, la transposition est immédiate et peut, sans problème, être automatisée.

Inversement, peut-on passer des doubles grammaires aux attributs sémantiques ? La réponse est beaucoup moins simple que précédemment car nous avons émis une critique sur les doubles grammaires qui est que dans une hyper-règle, on ne sait pas si une métanotion synthétise ou hérite de l'information. Il semble donc irrémédiable que, dans le cas général, la réponse soit négative. Mais si on se limite à des métanotions synthétisantes, la correspondance semble possible, en considérant chaque métanotion comme un attribut synthétisé. Nous pensons que cette question mérite une recherche approfondie car elle offrirait un débouché analytique aux doubles grammaires qui ajouterait à leurs possibilités synthétiques naturelles. Le passage attributs vers

doubles grammaires offre, quant à lui, à la méthode des attributs sémantiques, des possibilités synthétiques qu'elle n'a probablement pas. Cet aspect mérite, lui aussi, une étude approfondie. On peut établir le schéma de correspondance



### 8.3.3. *W-grammaires et grammaires affixes.*

Nous avons déjà souligné certaines similitudes entre les deux méthodes, au moins en ce qui concerne le métalangage syntaxique utilisé. En fait, la similitude est plus profonde. Considérons la définition du prédicat CHAINE en grammaire affixe (§6.3.3.).

CHAINE+L+M : (ISLET+L, TEST+L+M ; PASLET+L).

Les quatre occurrences de l'affixe L représentent la même notion et on doit rapprocher ce fait de la règle du remplacement uniforme des métanotations en doubles grammaires. Mais il s'agit ici d'une notion informatique car, en fait, ces occurrences accèdent à la même cellule mémoire alors qu'en doubles grammaires, il s'agit d'une notion linguistique, le langage engendré par une métanotation. Alors qu'en doubles grammaires, la notion terminale dérivée d'une métanotation est la même et reste la même pour toutes les métanotations d'une hyper-règle, en grammaires affixes, c'est l'adresse de la cellule mémoire qui reste la même, mais pas nécessairement l'information située à cette adresse. En fait, les grammaires affixes sont une version algorithmique et informatique simplifiée des possibilités linguistiques des doubles grammaires.

Peut-on passer d'une grammaire affixe à une double grammaire équivalente ? Cette transformation est triviale, en transformant les affixes en métanotions et en transformant les prédicats et actions en prédicats de doubles grammaires.

Prenons l'exemple de la déclaration des symboles de ruban de TURINGOL par grammaires affixes (§6.3.3.).

On a       DECL + T - D : TAPE, DCL + D, SDCL + D + T.  
               SDCL + X + Y - D : (VIRG, DCL + D, TEST + D + X,  
                                   FAIRE + Y + X + D, SDCL + Y + X ;  
                                   FIN + Y + X).

On obtient :

```
decl TAPE1           : tape symbol, dcl DEFT, sdcl DEFT TAPE1.
sdcl TAPE1 TAPE2     : virg symbol, dcl DEFT, test TAPE1 DEFT,
                     faire TAPE2 TAPE1 DEFT,
                     sdcl TAPE2 TAPE2 ;
                     fin TAPE2 TAPE1.
```

test TAPE DEFT : ou DEFT n'est pas dans TAPE.

faire TAPE1 TAPE2 DEFT : ou TAPE1 est TAPE2 DEFT.

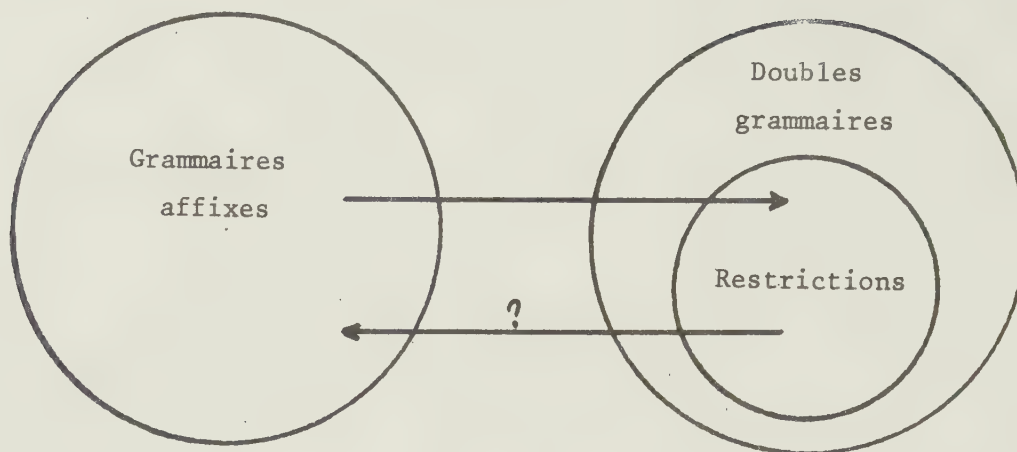
fin TAPE1 TAPE2 : ou TAPE1 est TAPE2.

Cette définition est incomplète et prend, là aussi, quelques libertés avec la syntaxe des doubles grammaires pour bien montrer la transformation.

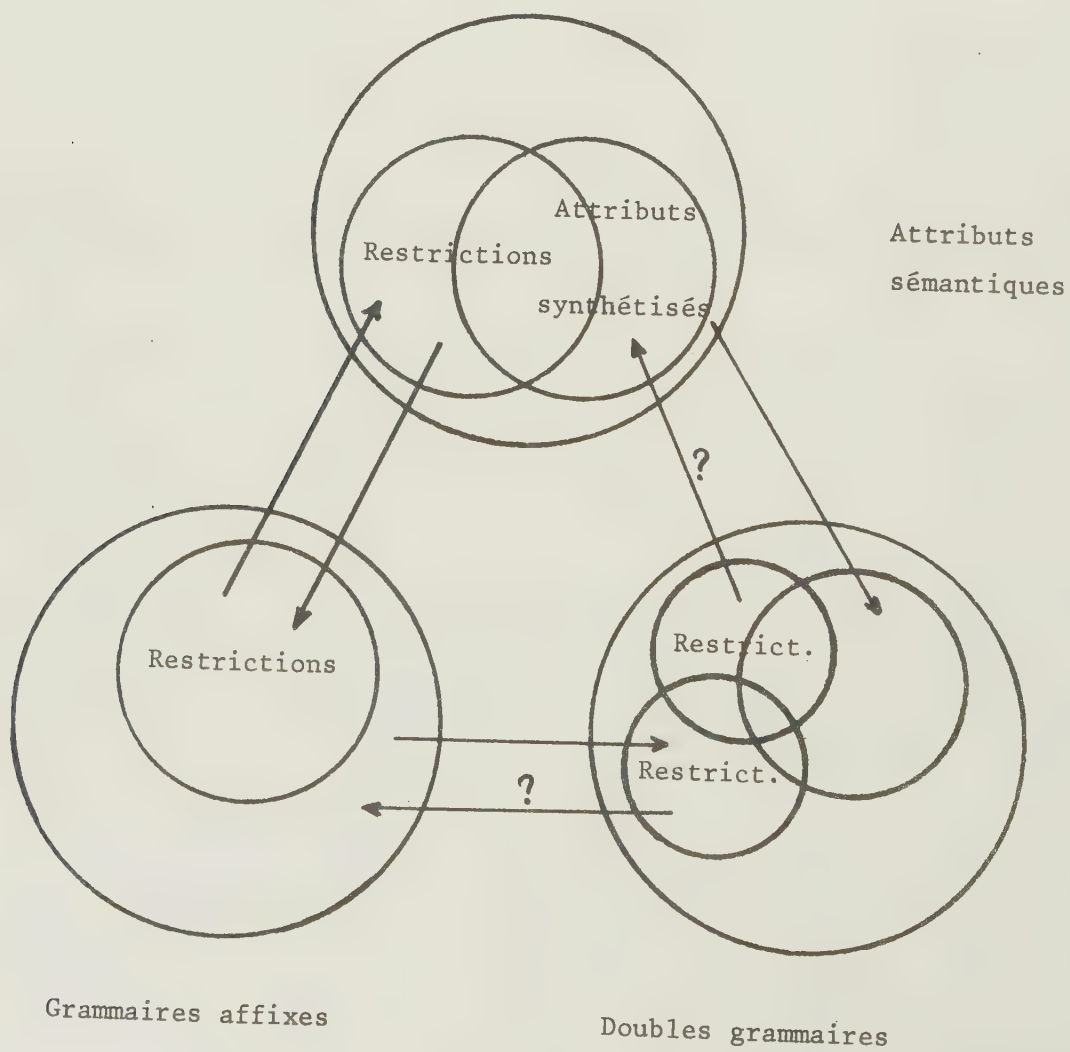
Inversement, peut-on passer de doubles grammaires en grammaires affixes ? La réponse est, en général, certainement négative. Il suffit de prendre l'exemple de la double grammaire obtenue en plaçant les symboles de ruban en fin de programme TURINGOL. On ne peut pas traiter cet exemple par une grammaire affixe.

Malgré tout, une recherche attentive s'impose pour voir sous quelles conditions on peut éventuellement, envisager cette transformation, encore une fois pour améliorer les possibilités analytiques des doubles grammaires qui en manquent singulièrement.

Nous avons le schéma de correspondance



#### 8.4. Schéma général de correspondance



## Deuxième Partie

METHODE DES ATTRIBUTS SEMANTIQUES : SYSTEME DELTA



## 9. NOTATIONS SYNTAXIQUES ET SEMANTIQUES, NOTION DE COHERENCE D'UN SYSTEME D'EQUATIONS.

### 9.1. Introduction.

Ce chapitre introduit des notations précises sur la définition de l'ensemble des règles d'écriture et sur la définition des attributs sémantiques. La notion de dépendance est ensuite définie ainsi que la notion de décoration d'un arbre syntaxique. La cohérence d'un système d'équations est établie.

Enfin, un rapprochement est fait entre la résolution d'un système d'équations et l'exécution de programmes écrits en Langages à Assignment Unique.

### 9.2. Notations et définitions syntaxiques.

Une grammaire hors contexte est définie par

$$G = \{N, T, Z, P\}$$

où  $N$  est l'ensemble fini des symboles non terminaux,

$T$  est l'ensemble fini des symboles terminaux,

$Z \in N$  est l'axiome de la grammaire

$P$  est l'ensemble fini des règles de productions.

. Nous notons l'ensemble  $P$  par un système de coordonnées [46]

$$\{X_{ij0} \rightarrow X_{ij1} \dots X_{ijk} \dots X_{ijn_{ij}}\}$$

pour  $0 \leq i \leq m$

$$1 \leq j \leq p_i$$

$$0 \leq k \leq n_{ij}$$

Les coordonnées  $(i, j, k)$  désignent l'occurrence d'un symbole  $X \in N \cup T \cup \{\epsilon\}$  dans la définition du  $i^e$  symbole non terminal, dans la  $j^e$  alternance, en position  $k$  dans cette alternance.

.  $X_{ij0}$  représente le symbole  $X_i$ .

.  $p_i (> 0)$  représente le nombre d'alternances de la définition de  $X_i$ .

.  $n_{ij} (> 0)$  représente le nombre de symboles dans la  $j^e$  alternance de la définition du symbole  $X_i$ . Dans ce chapitre et le chapitre 10, pour alléger

les notations, nous noterons  $n$  au lieu de  $n_{ij}$  dès qu'il n'y aura aucune ambiguïté, c'est-à-dire dès que  $i$  et  $j$  seront déterminés par le contexte.

- .  $X_0 = Z$  est l'axiome.
- . on suppose la grammaire  $G$  réduite, c'est à dire que tous les symboles non déterminaux sont accessibles et productifs.
- . à chaque phrase du langage décrit par  $G$ , on associe un arbre syntaxique ([26]) qui visualise la structure syntaxique de la phrase. On ne s'intéresse ici qu'à des arbres syntaxiques terminaux, c'est-à-dire dont les feuilles sont étiquetées par des symboles terminaux.
- . on appelle hauteur d'un arbre syntaxique le nombre de branches du plus long chemin dans l'arbre entre sa racine et ses feuilles.

### 9.3. Notations et définitions sémantiques.

#### 9.3.1. *Définitions préliminaires.*

- . à chaque symbole  $X \in N$ , on associe deux ensembles finis,  $H(X)$  et  $S(X)$  représentant les attributs hérités et synthétisés de  $X$ .
- . pour tout  $X \in N$ ,  
on impose  $H(X) \cap S(X) = \emptyset$   
on note  $A(X) = H(X) \cup S(X)$
- . à tout  $X \in T$ , on associe un ensemble  $S(X)$  composé du seul attribut standard text qui permet d'accéder au texte symbolique du symbole terminal  $X$ .

#### 9.3.2. *Notations et définitions.*

Considérons une règle de la forme

$$X_{ij0} \rightarrow X_{ij1} \dots X_{ijn}$$

Nous allons associer à cette règle des fonctions sémantiques qui définissent les attributs des symboles de cette règle. En réalité, nous verrons que ces fonctions définissent des occurrences d'attributs liés aux noeuds du sous arbre syntaxique associé à cette règle.

Pour une application donnée d'une règle de coordonnées  $(i,j)$ , une occurrence d'attributs peut être caractérisée par le couple d'un attribut  $a$  et d'un entier  $k$  avec  $0 \leq k \leq n$  et  $a \in A(X_{ijk})$ . Elle sera fonction d'autres occurrences d'attributs de la même application de règle si

$$a \in S(X_{ij0}) \text{ et } k = 0$$

$$\text{ou } a \in H(X_{ijk}) \text{ et } 1 \leq k \leq n$$

Pour exprimer le calcul de l'occurrence de l'attribut  $(a,k)$  en fonction d'autres occurrences d'attributs  $(a_1, l_1), \dots, (a_t, l_t)$ , on introduit une fonction  $f_{ijka}$ .

En résumé, pour toute règle de coordonnées  $(i,j)$ , on associe à tout couple  $(a,k)$  vérifiant les conditions précédentes une fonction  $f_{ijka}$  de  $t(i,j,k,a)^*$  variables, et un profil formé de  $t$  couples  $(a_1, l_1), \dots, (a_t, l_t)$  avec  $0 \leq l_x \leq n$  et  $a_x \in A(X_{ijk l_x})$ , pour  $1 \leq x \leq t$ .

Ces fonctions permettent d'associer à chaque règle, un système d'équations :

$$SE(i,j) = \{(a,k) = f_{ijka}((a_1, l_1), \dots, (a_t, l_t))\}$$

que nous appellerons système générique de la règle  $(i,j)$ .

$$\text{On pose } EA(i,j) = \{(a,k) \mid a \in A(X_{ijk}) \text{ et } 0 \leq k \leq n\}.$$

C'est l'ensemble des attributs associés à la règle  $(i,j)$ .

$$\text{On pose } AD(i,j) = \{(a,k) \mid (a,k) \in EA(i,j) \text{ et } f_{ijka} \text{ existe}\}.$$

C'est l'ensemble des attributs définis dans la règle  $(i,j)$ .

On définit une relation de dépendance  $DE(i,j)$  dans  $EA(i,j)$  par  $(a,k) DE(i,j) (b,l)$  si et seulement si  $(b,l)$  est un des couples du profil de  $f_{ijka}$ .

Cette relation exprime les dépendances élémentaires entre les attributs de la règle  $(i,j)$ . Elle peut se lire : l'attribut  $(a,k)$  DEpend de l'attribut  $(b,l)$ .

On définit un graphe de dépendance associé au système  $SE(i,j)$  par l'ensemble des relations binaires du graphe  $DE(i,j)^{-1}$ .

On note simplement  $(a,k) \leftarrow (b,l)$  pour  $(a,k) DE(i,j) (b,l)$ .

---

\* Nous noterons brièvement  $t$  au lieu de  $t(i,j,k,a)$ .

#### 9.4. Exemples.

Nous allons illustrer toutes ces définitions à l'aide d'une règle de la définition de l'algorithme de Sethu-Ullman (§7.2).

```

<expression> ::= <expression> <pm> <opérande>
g(expression') := vrai
g(opérande)     := faux
r(expression)   := si r(expression') = r(opérande)
                  alors      r(expression') + 1
                  sinon      max(r(expression'), r(opérande))

```

Notations en coordonnées :

$$X_{220} \longrightarrow X_{221} \quad X_{222} \cdot X_{223}$$

Système d'équations :

$$SE(2,2) = \begin{cases} (g,1) = \text{vrai} \\ (g,3) = \text{faux} \\ (r,0) = \text{si } (r,1) = (r,3) \text{ alors } (r,1) + 1 \\ \quad \text{sinon } \max((r,1), (r,3)). \end{cases}$$

Ensemble d'attributs :

$$EA(2,2) = \{(g,1), (g,3), (r,0), (r,1), (r,3)\}$$

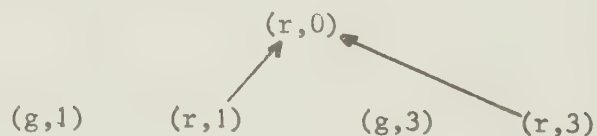
$$AD(2,2) = \{(g,1), (g,3), (r,0)\}$$

Dépendance élémentaire

$$(r,0) \text{ DE}(2,2) (r,1)$$

$$(r,0) \text{ DE}(2,2) (r,3)$$

Graphe de dépendance



### 9.5. Notion de décoration d'un arbre syntaxique.

#### 9.5.1. *Présentation.*

Chaque système d'équation  $SE(i,j)$  du paragraphe précédent est le système générique de la règle  $(i,j)$ .

L'opération de décoration va consister à transformer les systèmes génériques en systèmes effectifs, lors de la construction de l'arbre syntaxique associé à chaque règle.

#### 9.5.2. *Opération de décoration.*

Soit un arbre syntaxique. Numérotons ses noeuds, de manière injective, par des entiers  $1, 2, \dots, z$ , de telle sorte que l'on puisse distinguer les occurrences des attributs aux noeuds de l'arbre, comme des couples  $(a, x)$ , où  $a$  est un attribut de l'étiquette du noeud numéroté  $x$ .

Nous allons associer à l'arbre syntaxique un système d'équations dont les inconnues sont ces couples  $(a, x)$ . Ce système peut être défini par récurrence sur l'arbre syntaxique :

i) si l'arbre se réduit à un noeud numéroté  $x$ , étiqueté par un symbole terminal  $X$ , le système est réduit à une seule équation

$$(text, x) = X.$$

ii) si l'arbre n'est pas réduit à une feuille, il s'écrit (cf § 1.4.2.)

$$r = X_{ij0} \times (r_1 + \dots + r_n)$$

ce qui signifie qu'il est composé d'une racine  $X_{ij0}$  et de  $n$  arbres syntaxiques  $r_1, \dots, r_n$  de racines respectives  $X_{ij1}, \dots, X_{ijn}$ , pour une règle  $(i,j)$ .

Soient  $x_0, x_1, \dots, x_n$  les numéros des racines de  $r, r_1, \dots, r_n$ .

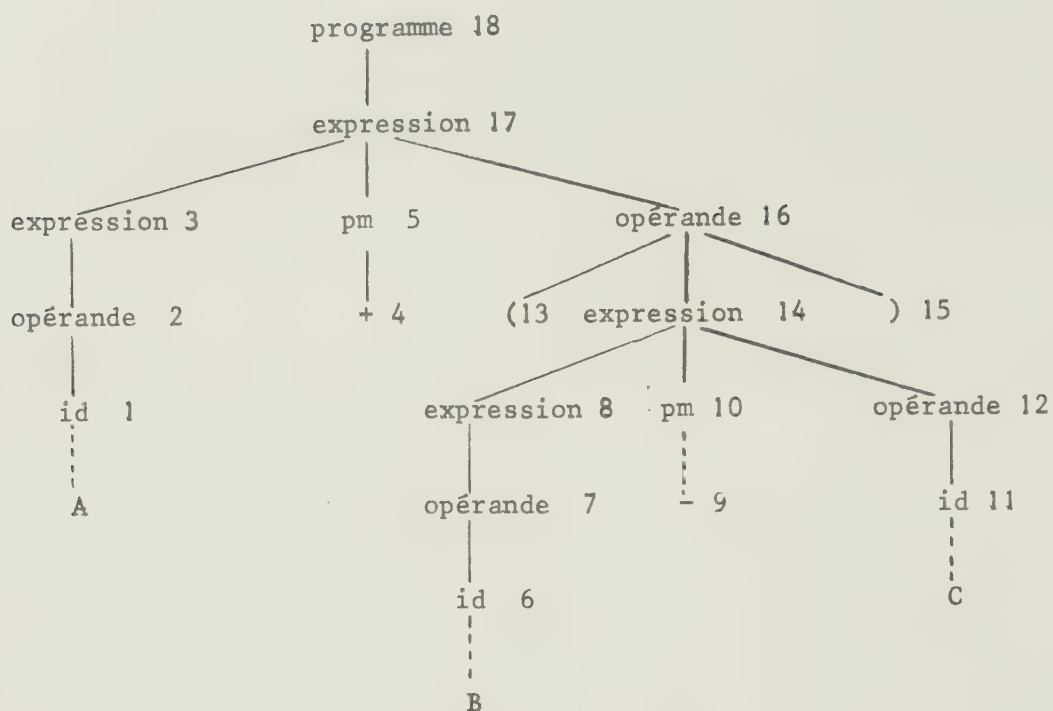
Le système associé à  $r$  est formé en réunissant les systèmes associés à  $r_1, \dots, r_n$  et les équations  $(a, x_k) = f_{ijka}((a_1, x_{11}), \dots, (a_t, x_{1t}))$  où  $f_{ijka}$  est une fonction de profil  $(a_1, l_1), \dots, (a_t, l_t)$  associée à la règle  $(i,j)$ .

#### 9.5.3. *Exemple.*

Continuons à prendre comme exemple l'algorithme de Sethi-Ullman et construisons le système effectif associé au programme

$$A + (B - C)$$

L'arbre syntaxique numéroté est :



L'opération de décoration, appliquée à l'arbre numéroté, donne le système effectif suivant, où les inconnues ne sont pas représentées sous la forme de couple (attribut, numéro de noeud) mais sous la forme plus traditionnelle : attribut indicé par le numéro de noeud.

noeud

$$18 \begin{cases} r_{18} = r_{17} \\ g_{17} = \text{vrai} \end{cases}$$

$$17 \begin{cases} r_{17} = \text{si } r_3 = r_{16} \text{ alors } r_3 + 1 \text{ sinon } \max(r_3, r_6) \\ g_3 = \text{vrai} \\ g_{16} = \text{faux} \end{cases}$$

$$3 \begin{cases} r_3 = r_2 \\ g_2 = g_3 \end{cases}$$

$$2 \quad r_2 = \text{si } g_2 \text{ alors } 1 \text{ sinon } 0$$

$$1 \quad \text{text}_1 = 'A'$$

$$4 \quad \text{text}_4 = '+'$$

$$16 \begin{cases} r_{16} = r_{14} \\ g_{14} = g_{16} \end{cases}$$

$$13 \quad \text{text}_{13} = '{'$$

$$15 \quad \text{text}_{15} = ')'$$

$$14 \begin{cases} r_{14} = \text{si } r_8 = r_{12} \text{ alors } r_8 + 1 \text{ sinon } \max(r_8, r_{12}) \\ g_8 = \text{vrai} \\ g_{12} = \text{faux} \end{cases}$$

$$8 \begin{cases} r_8 = r_7 \\ g_7 = g_8 \end{cases}$$

$$7 \quad r_7 = \text{si } g_7 \text{ alors } 1 \text{ sinon } 0$$

$$6 \quad \text{text}_6 = 'B'$$

$$9 \quad \text{text}_9 = '-'$$

$$12 \quad r_{12} = \text{si } g_{12} \text{ alors } 1 \text{ sinon } 0$$

$$11 \quad \text{text}_{11} = 'C'$$



Notons  $D(S)$  la fermeture transitive stricte de la relation  $DE(S)$ , c'est à dire la plus petite relation transitive contenant  $DE(S)$ .

$$D(S) = DE(S)^+$$

Pour le chapitre suivant, nous avons besoin d'établir un résultat sur les relations de dépendance liées à un système d'équations. Plus généralement, considérons  $n$  relations binaires notées  $R_1, \dots, R_n$  dans un même ensemble.

*Proposition 1 :*  $(R_1 \vee R_2 \vee \dots \vee R_n)^+ = (R_1^+ \vee R_2^+ \vee \dots \vee R_n^+)^+$

Démonstration :

i)  $(R_1^+ \vee R_2^+ \vee \dots \vee R_n^+)^+$  est la plus petite relation transitive contenant

$$R_1, R_2, \dots, R_n$$

ii)  $(R_1^+ \vee R_2^+ \vee \dots \vee R_n^+)^+$  est la plus petite relation transitive contenant

$$R_1^+, R_2^+, \dots, R_n^+$$

Comme une relation transitive qui contient  $R_i$  contient aussi  $R_i^+$  et réciproquement, il y a égalité entre les deux relations transitives précédentes i) et ii).

Revenons à un système d'équations : si un tel système est obtenu par réunion de  $n$  sous systèmes  $S_1, S_2, \dots, S_n$ , et si nous notons  $DE(S_i)$  la relation de dépendance élémentaire entre les inconnues définies dans  $S_i$ , donc appartenant à  $S$ , et les inconnues de  $S$  qui permettent de les calculer, on a naturellement

$$DE(S) = DE(S_1) \vee DE(S_2) \vee \dots \vee DE(S_n)$$

Comme corollaire de la proposition 1, nous avons alors :

$$D(S) = (D(S_1) \vee D(S_2) \vee \dots \vee D(S_n))^+$$

### 9.7. Cohérence d'un système d'équations.

On dit qu'un système d'équations est cohérent s'il admet une solution unique.

Nous allons établir une condition de cohérence, à partir de la relation  $DE(S)$  ou, pour faciliter la présentation, à partir du graphe de dépendance lié à un système.

Supposons que ce graphe est sans circuit :

On peut alors ordonner les inconnues de manière que si  $x_i DE(S)x_j$  alors  $x_j$  précède  $x_i$ . Il en résulte que le système admet une solution unique obtenue en calculant les inconnues dans l'ordre ainsi déterminé.

Pour déterminer si le graphe est sans circuit, il suffit de calculer la relation  $D(S) = DE(S)^+$  introduite au paragraphe précédent.

On a donc le théorème suivant :

*Théorème 2* : un système (S) d'équations est cohérent si aucune de ses inconnues  $x_i$  ne vérifie  $x_i D(S) x_i$ .

On doit noter que cette condition de cohérence n'est pas nécessaire, dans le cas le plus général.

Par exemple, l'équation  $x = 2x + 1$  admet une solution dans l'ensemble des réels avec  $x \in DE(S)$ .

Cette condition devient nécessaire si on impose une solution unique quelle que soit l'interprétation des opérateurs du système.

Il faut remarquer que si la démonstration du théorème 2 introduit une méthode de résolution par ordonnancement des inconnues, cela n'impose aucunement qu'on résolve le système par ce procédé. Par exemple, on pourrait également procéder par substitution.

#### 9.8. Langages à assignation unique (LAU) ([50] , [51])

Comme leur nom l'indique, ces langages n'autorisent qu'une seule affectation à chaque variable du programme (cette restriction est modulée pour les variables apparaissant dans les boucles, bien entendu, mais ce point ne nous intéresse pas ici).

L'intérêt de tels langages n'est pas évident mais un exemple simple va l'éclairer.

Considérons le programme suivant :

$g_3 = \text{vrai}$

$g_2 = g_3$

$r_2 = \text{si } g_2 \text{ alors } 1 \text{ sinon } 0$

$r_3 = r_2$

$g_8 = \text{vrai}$

$g_7 = g_8$

$r_7 = \text{si } g_7 \text{ alors } 1 \text{ sinon } 0$

$r_8 = r_7$

$g_{12} = \text{faux}$

$r_{12} = \text{si } g_{12} \text{ alors } 1 \text{ sinon } 0$

```

r14 = si r8 = r12 alors r8 + 1 sinon max(r8, r12)
r16 = r14
r17 = si r3 = r16 alors r3 + 1 sinon max(r3, r16)
r18 = r17

```

Ce programme respecte la contrainte des LAU et il est exécutable séquentiellement, de la première à la dernière instruction.

Maintenant, appliquons une permutation quelconque aux instructions de ce programme : le programme obtenu ne peut plus s'exécuter séquentiellement mais sa sémantique est restée invariante.

En effet, la règle de l'assignation unique permet d'affirmer qu'une instruction, où qu'elle soit située dans le programme, ne peut s'exécuter que lorsque tous ses opérandes sont évalués et connus ; or chacun d'entre eux est défini exactement une fois.

### 9.9. Système d'équations, langages à assignation unique .

Le lien entre les systèmes d'équations mathématiques - et les systèmes d'équations sémantiques en font naturellement partie - et les langages à assignation unique est naturel et évident : les systèmes d'équations sont des programmes LAU.

Résolution mathématique d'un système, évaluation sémantique des équations sémantiques et exécution de programmes LAU constituent le même type de traitement. En particulier, les conditions de cohérence introduites au paragraphe 9.7. sont des conditions suffisantes d'exécution correcte des programmes LAU. Inversement, certains résultats sur l'exécution des programmes LAU peuvent être exploités pour la résolution des systèmes d'équations. En particulier, les possibilités d'exécution en parallèle sont les plus importantes pour le développement futur de la résolution informatique des systèmes d'équations et donc des méthodes par attributs sémantiques. Nous y reviendrons aux chapitres 11 et 12.

## 10. COHERENCE D'UNE DEFINITION PAR ATTRIBUTS SEMANTIQUES.

### 10.1. Introduction.

Ce chapitre a pour but de définir la notion de cohérence d'une définition et de donner un certain nombre d'algorithmes de tests de cohérence.

### 10.2. Notion de cohérence.

Définition : une définition par attributs sémantiques est cohérente si le système d'équations sémantiques obtenu par décoration d'un arbre syntaxique quelconque de racine étiquetée par l'axiome de la grammaire, admet une solution unique.

On voit tout l'intérêt de cette notion. Si on peut prouver la cohérence d'une définition, cela implique que la résolution de tout système d'équations sémantiques, ou évaluation sémantique, est possible.

Si on se souvient des doutes permanents que laissent planer les méthodes traductives du chapitre 6, on se rend compte de l'importance, tant d'un point de vue théorique que d'un point de vue pratique, de cette notion.

Pour établir des conditions de cohérence, **nous** allons transposer sur la définition sémantique, les conditions établies au chapitre précédent sur les équations sémantiques.

### 10.3. Définition unique.

La définition de la cohérence impose que chaque inconnue ait une solution unique. Il nous faut donc imposer que chaque attribut est défini de façon unique. Cette condition est, théoriquement, sans objet mais doit être incorporée à des algorithmes de preuve de la cohérence d'une définition sémantique.

i) attributs synthétisés :

$$\forall i \quad 0 \leq i \leq m$$

$$\forall a \in S(X_i)$$

$$\forall j \quad 1 \leq j \leq p_i$$

la définition sémantique contient une fonction unique  $f_{i,j,a}$

Ces conditions imposent

1. la définition des attributs synthétisés des symboles dans toutes leurs alternances de définition,
2. l'unicité de cette définition.

ii) attributs hérités :

$$\begin{aligned}
 &\forall p \quad 0 \leq p \leq m \\
 &\forall a \in H(X_p) \\
 &\forall i \quad 0 \leq i \leq m \\
 &\quad \forall j \quad 1 \leq j \leq p_i \\
 &\quad \quad \forall k \quad 1 \leq k \leq n \\
 &\quad \quad X_{ijk} = X_p \implies
 \end{aligned}$$

la définition sémantique contient une fonction unique  $f_{ijka}$

Ces conditions imposent :

1. la définition des attributs hérités de toutes les occurrences des symboles en partie droite d'une règle.
2. l'unicité de cette définition.

#### 10.4. Non circularité.

##### 10.4.1. Définitions.

Rappelons la condition de non circularité sur le système  $S$  d'équations associé à un arbre syntaxique  $r$  par l'opération de décoration :

*condition* :  $\forall X_i \in N, \forall r$  arbre syntaxique de racine  $X_i$  numéroté  $x$ ,  
 $\forall a \in A(X_i)$

$$\text{non } (a, x) \in D(S) (a, x)$$

Plus généralement, nous voulons exprimer les dépendances entre attributs des noeuds d'un arbre  $r$  de racine  $X_i$ .

On définit  $D(r)$  par

$$a \in D(r) b \iff a \in A(X_i) \text{ et } b \in A(X_i) \text{ et } (a, x) \in D(S) (b, x)$$

La condition de non circularité s'écrit alors :

*condition* :  $\forall X_i \in N, \forall r$  arbre syntaxique de racine  $X_i, \forall a \in A(X_i)$

$$\text{non } a \in D(r) a$$

Mais nous voulons établir la condition sur la définition en général.

On définit  $D(X_i) = \bigvee_r D(r) \mid r$  arbre syntaxique de racine  $X_i$ .

La condition de non circularité s'écrit alors :

condition :  $\forall X_i \in N, \forall a \in A(X_i) \quad \text{non } a D(X_i) a$

Pour toute relation B entre attributs, appelons  $\phi_k(B)$  la relation définie par :

$$(a,k) \phi_k(B) (b,k) \iff a B b$$

Inversement, étant donné une relation B entre couples (a, k), on note pr(B), pour projection de B, la relation définie par :

$$a \text{ pr}(B) b \iff \exists k, \exists l \quad (a,k) B (b,l)$$

Considérons un arbre syntaxique r de racine  $X_{ijo}$ . Il s'écrit :

$$r = X_{ijo} \times (r_1 + \dots + r_n)$$

D'après la définition de D(r), la définition du système d'équations associé à r et la proposition 1, on obtient :

$$D(r) = \text{pr}(\phi_1(D(r_1)) \vee \dots \vee \phi_n(D(r_n)) \vee DE(i,j))^+$$

Et d'après la définition de  $D(X_i)$ , on a :

$$D(X_i) = \bigvee_j \text{pr}(\phi_1(D(r_1)) \vee \dots \vee \phi_n(D(r_n)) \vee DE(i,j))^+ \\ \text{pour } 1 \leq j \leq p_i \text{ et } \forall k \ 1 \leq k \leq n \ D(r_k) \in D(X_{ijk})$$

Cette formulation de  $D(X_i)$  est à la base des algorithmes de tests de circularité des paragraphes suivants.

#### 10.4.2. Exemples.

Nous allons considérer deux définitions par attributs sémantiques qui servent d'exemples aux algorithmes de tests de circularité.

##### 10.4.2.1. Exemple 1 :

grammaire

$$A \rightarrow B$$

$$B \rightarrow x$$

$$B \rightarrow y$$

dépendances élémentaires

$$(a,0) DE(1,1) (d,0)$$

$$(b,0) DE(1,1) (b,1)$$

$$(c,0) DE(1,1) (b,0)$$

$$(d,0) DE(1,1) (d,1)$$

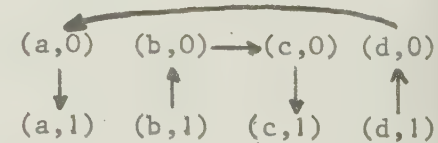
$$(a,1) DE(1,1) (a,0)$$

$$(c,1) DE(1,1) (c,0)$$

$$(b,0) DE(2,1) (a,0)$$

$$(d,0) DE(2,2) (c,0)$$

graphe de dépendance



$$(a,0) \rightarrow (b,0) \quad (c,0) \rightarrow (d,0)$$

$$(a,0) \rightarrow (b,0) \quad (c,0) \rightarrow (d,0)$$

## 10.4.2.2. Exemple 2 :

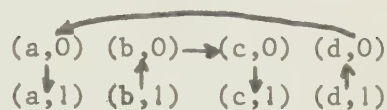
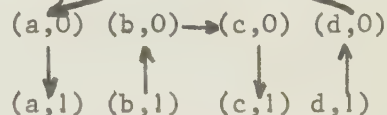
grammaire

 $A \longrightarrow B$ 

dépendances élémentaires

 $(a,0) \text{ DE}(1,1) (d,0)$  $(b,0) \text{ DE}(1,1) (b,1)$  $(c,0) \text{ DE}(1,1) (b,0)$  $(d,0) \text{ DE}(1,1) (d,1)$  $(a,1) \text{ DE}(1,1) (a,0)$  $(c,1) \text{ DE}(1,1) (c,0)$ 

graphe de dépendance

 $A \longrightarrow C$  $(a,0) \text{ DE}(1,2) (d,0)$  $(b,0) \text{ DE}(1,2) (b,1)$  $(c,0) \text{ DE}(1,2) (b,0)$  $(d,0) \text{ DE}(1,2) (d,1)$  $(a,1) \text{ DE}(1,2) (a,0)$  $(c,1) \text{ DE}(1,2) (c,0)$  $B \longrightarrow x$  $(b,0) \text{ DE}(2,1) (a,0)$  $(a,0) \longrightarrow (b,0) \quad (c,0) \quad (d,0)$  $C \longrightarrow y$  $(d,0) \text{ DE}(3,1) (c,0)$  $(a,0) \quad (b,0) \quad (c,0) \longrightarrow (d,0)$ 10.5. Premier algorithme (A1)10.5.1. *Algorithme.*

Le premier algorithme découle directement de la définition de  $D(X_i)$  du § 10.4.1.

$$D(X_i) = \bigvee_j \text{pr}(\phi_1(D(r_1)) \vee \dots \vee \phi_n(D(r_n)) \vee \text{DE}(i,j))^+$$

pour  $1 \leq j \leq p_i$  et  $\forall k \quad 1 \leq k \leq n \quad D(r_k) \in D(X_{ijk})$

On obtient un système d'équations ensemblistes d'inconnues  $D(X_0), D(X_1), \dots, D(X_m)$ , qu'il nous faut résoudre.

Nous allons résoudre ce système par approximations successives en considérant la hauteur des arbres syntaxiques (§ 9.2).

Pour cela, posons :

$$D_p(X_i) = \bigvee_r D(r) \quad \left| \begin{array}{l} r \text{ arbre syntaxique de racine } X_i \\ \text{et de hauteur } \leq p. \end{array} \right.$$

On a alors :

si  $X_i \in T$   $D_p(X_i) = \{\phi\}$   $p \geq 0$  , où  $\phi$  désigne la relation vide

si  $X_i \in N$

$D_0(X_i) = \phi$  car il n'existe pas d'arbre syntaxique de racine  $X_i$  et de hauteur nulle

$$D_p(X_i) = \bigvee_j \text{pr}(\phi_1(D(r_1)) \vee \dots \vee \phi_n(D(r_n)) \vee DE(i,j))^+ \\ \text{pour } 1 \leq j \leq p_i \text{ et } \forall k \ 1 \leq k \leq n \ D(r_k) \in D_{p-1}(X_{ijk})$$

La convergence de ces approximations successives est assurée par le nombre fini de relations entre attributs.

La solution est obtenue dès que

$$\forall i \ 0 \leq i \leq m \quad D_{p+1}(X_i) = D_p(X_i)$$

Alors,  $D(X_i) = D_p(X_i)$

La condition de non circularité s'écrit alors

condition :  $\forall i \ 0 \leq i \leq m, \ \forall a \in A(X_i), \ \forall D(r) \in D(X_i)$   
non a  $D(r)$  a

### 10.5.2. Exemple.

Considérons l'exemple 1 (§ 10.4.2.1.).

On obtient :

$$D_p(x) = D_p(y) = \{\phi\} \quad p \geq 0$$

$$D_0(A) = \phi$$

$$D_0(B) = \phi$$

$$D_1(A) = \phi$$

$$D_1(B) = \{a \rightarrow b \ c \ d, \ a \ b \ c \rightarrow d\}$$

$$D_2(A) = \{a \rightarrow b \rightarrow c \rightarrow d, \ a \leftarrow b \leftarrow c \leftarrow d\}$$

$$D_2(B) = D_1(B)$$

$$D_3(A) = D_2(A)$$

$$D_3(B) = D_2(B)$$

La convergence est atteinte : il n'y a aucune circularité de définition.

### 10.6. Deuxième algorithme.

#### 10.6.1. *Première version (A21)*

##### 10.6.1.1. Algorithme.

On pourrait penser qu'il est superflu de conserver l'ensemble des relations associées à un symbole et qu'on pourrait se contenter de définir une nouvelle relation  $R(X_i)$  définie par

$$R(X_i) = \bigvee_j \text{pr}(\phi_j(R(X_{ij1})) \vee \dots \vee \phi_n(R(X_{ijn})) \vee \text{DE}(i,j))^+ \\ \text{pour } 1 \leq j \leq p_i$$

On obtient ainsi un système d'équations entre relations, système que l'on résout également par approximations successives.

On a alors :

$$\text{si } X_i \in T \quad R_p(X_i) = \phi \quad p \geq 0$$

$$\text{si } X_i \in N$$

$$R_0(X_i) = \phi$$

$$R_p(X_i) = \bigvee_j \text{pr}(\phi_j(R_{p-1}(X_{ij1})) \vee \dots \vee \phi_n(R_{p-1}(X_{ijn})) \vee \text{DE}(i,j))^+ \\ \text{pour } 1 \leq j \leq p_i$$

Comparons cet algorithme au premier algorithme. Pour cela, montrons, par récurrence sur  $p$ , la proposition 3 :

*proposition 3 :*

$$D(r) \in D_p(X_i) \implies D(r) \subset R_p(X_i)$$

( $\subset$  désigne, ici, l'inclusion de relations).

- i) pour  $p = 0$  évident
- ii) supposons la proposition vraie jusqu'à  $p-1$   
démontrons la pour  $p$

$D(r) \in D_p(X_i)$  donc, par définition de  $D_p(X_i)$ , on a

$$D(r) = \text{pr}(\phi_j(D(r_j)) \vee \dots \vee \phi_n(D(r_n)) \vee \text{DE}(i,j))^+ \text{ pour } j \text{ donné}$$

$$\text{et avec } \forall k \quad 1 \leq k \leq n \quad D(r_k) \in D_{p-1}(X_{ijk})$$

mais, par l'hypothèse de récurrence, on a

$$\forall k \quad 1 \leq k \leq n \quad D(r_k) \in D_{p-1}(X_{ijk}) \implies D(r_k) \subset R_{p-1}(X_{ijk})$$

$$\text{donc} \quad \bigvee_k \phi_k(D(r_k)) \subset \bigvee_k \phi_k(R_{p-1}(X_{ijk}))$$

$$\text{donc} \quad \text{pr}((\bigvee_k \phi_k(D(r_k))) \vee DE(i,j))^+ \subset \text{pr}((\bigvee_k \phi_k(R_{p-1}(X_{ijk}))) \vee DE(i,j))^+$$

$$\text{donc} \quad D(r) \subset R_p(X_{ijk})$$

La proposition 3 est démontrée.

Mais, en général, la réciproque est fausse. On en conclut que, s'il y a une circularité, elle est détectée par A1 et A21, mais si A21 trouve une circularité, rien ne dit qu'il y en a effectivement une.

#### 10.6.1.2. Exemple 1 :

Les approximations successives donnent

$$R_0(A) = a \quad b \quad c \quad d$$

$$R_0(B) = a \quad b \quad c \quad d$$

$$R_1(A) = a \quad b \quad c \quad d$$

$$R_1(B) = a \rightarrow b \quad c \rightarrow d$$

$$R_2(A) = a \rightarrow b \rightarrow c \rightarrow d \quad (\text{simplifiée})$$

$$R_2(B) = R_1(B)$$

$$R_3(A) = R_2(A)$$

$$R_3(B) = R_2(B)$$

La convergence est atteinte et l'algorithme A21 trouve une circularité qui n'existe pas, en fait, d'après A1.

## 10.6.1.3. Exemple 2 :

On obtient successivement

$$R_0(A) = a \ b \ c \ d$$

$$R_0(B) = a \ b \ c \ d$$

$$R_0(C) = a \ b \ c \ d$$

$$R_1(A) = a \ b \ c \ d$$

$$R_1(B) = a \rightarrow b \ c \ d$$

$$R_1(C) = a \ b \ c \rightarrow d$$

$$R_2(A) = a \rightarrow b \rightarrow c \rightarrow d \quad (\text{simplifiée})$$

$$R_2(B) = R_1(B)$$

$$R_2(C) = R_1(C)$$

$$R_3(A) = R_2(A)$$

$$R_3(B) = R_2(B)$$

$$R_3(C) = R_2(C)$$

La convergence est atteinte et une circularité est détectée.

## 10.6.2. Deuxième version (A22)

## 10.6.2.1. Algorithme :

Nous allons modifier l'algorithme précédent pour associer une relation  $R$  non plus à un symbole mais à une règle.

Cet algorithme A22 est l'algorithme donné par KNUTH en [1a].

Posons

$$R(i,j) = \bigvee_{l_1, \dots, l_n} \text{pr}(\phi_1(R(r_1)) \vee \dots \vee \phi_n(R(r_n)) \vee \text{DE}(i,j))^+$$

$$\text{avec } \forall k \quad 1 \leq k \leq n, \text{ si } X_{ijk} = X_u \text{ alors } R(r_k) = R(u, l_k)$$

$$\text{pour } 1 \leq l_k \leq n_{l_k u}$$

On obtient toujours un système d'équations entre relations que l'on résout aussi par approximations successives.

$$R_0(i,j) = \phi$$

$$R_p(i,j) = \bigvee_{l_1, \dots, l_n} \text{pr}(\phi_1(R(r_1)) \vee \dots \vee \phi_n(R(r_n)) \vee \text{DE}(i,j))^+$$

$$\text{avec } \forall k \quad 1 \leq k \leq n, \text{ si } X_{ijk} = X_u \text{ alors } R(r_k) = R_{p-1}(u, l_k) \\ \text{pour } 1 \leq l_k \leq n_{l_k u}$$

On peut montrer la proposition 4

$$\text{proposition 4 : } D(r) \in D_p(X_i) \Rightarrow \exists j \quad 1 \leq j \leq n : D(r) \in R_p(i,j)(X_i)$$

La démonstration est similaire à la démonstration de la proposition 3.

Nous avons supposé la grammaire réduite, donc chaque symbole, à l'exception éventuelle de l'axiome, apparaît au moins une fois en partie droite d'une règle. Ce qui entraîne que

$$\forall i \quad 0 \leq i \leq m \quad R_p(X_i) = \bigvee_{j \quad 1 \leq j \leq p_i} R_{p-1}(i,j)$$

On peut alors réécrire la résolution du système comme :

$$R_0(i,j) = \phi$$

$$R_p(i,j) = \text{pr}(\phi_1(R_{p-1}(X_{ij1})) \vee \dots \vee \phi_n(R_{p-1}(X_{ijn})) \vee \text{DE}(i,j))^+$$

Une différence apparaît alors entre les deux algorithmes A2, pour l'axiome de la grammaire qui peut ne pas apparaître en partie droite, ce qui entraîne que l'on n'a pas à calculer  $R_p(Z)$  comme la réunion des  $R_p(0,j) \quad 1 \leq j \leq p_0$

$R(Z)$  peut alors trouver une circularité alors qu'aucun  $R(0,j)$  n'en met en évidence.

## 10.6.2.2. Exemple 1 :

Dans cet exemple, l'axiome A n'a qu'une définition donc les algorithmes A21 et A22 trouvent tous deux une circularité car

$$R(A) = R(0,1)$$

## 10.6.2.3. Exemple 2 :

Par application de l'algorithme A22, on obtient :

$$R(0,1) = a \rightarrow b \rightarrow c \rightarrow d$$

$$R(0,2) = a \rightarrow b \rightarrow c \rightarrow d$$

A22 ne détecte pas de circularité alors que A21 en a trouvé une car

$$R(A) = R(0,1) \vee R(0,2)$$

donc

$$R(A) = a \rightarrow b \rightarrow c \rightarrow d \quad (\text{simplifiée})$$

10.7. Troisième algorithme (A3)

L'algorithme A1 présente un sérieux inconvénient d'implantation : il résout le système ensembliste de relations globalement, c'est à dire, à l'itération  $p$  et pour chaque symbole, on doit mémoriser les ensembles  $D_{p-1}$  et  $D_p$ .

On peut accélérer la convergence en résolvant ce système, équation par équation, et non globalement. Ceci permet aussi de ne mémoriser qu'un seul ensemble de relations par symbole.

L'algorithme A3 se déduit de A1 par cette constatation.

pour  $i := 0$  pas 1 jusqu'à  $m$  faire  $D(X_i) = \phi$

jusqu'à convergence faire

pour  $i := 0$  pas 1 jusqu'à  $m$  faire

$$D(X_i) = \bigvee_j \text{pr}(\phi_j(D(r_1)) \vee \dots \vee \phi_n(D(r_n)) \vee DE(i,j))^+$$

pour  $1 \leq j \leq p_i$  et

$$\forall k \quad 1 \leq k \leq n \quad D(r_k) \in D(X_{ijk})$$

Cet algorithme est identique au deuxième algorithme de KNUTH [1b].

### 10.8. Quatrième algorithme (A4).

Nous allons modifier l'algorithme A3 à l'aide de deux résultats importants.

#### 10.8.1. Elimination des relations inutiles.

Dans tous les algorithmes de ce chapitre, nous sommes amenés à résoudre des systèmes d'équations. Mais nous nous intéressons moins à cette résolution qu'à la détection de circularité. Aussi, nous souhaitons ne conserver que les relations utiles à cette détection.

Pour cela, considérons deux ensembles de relations A et B.

On définit que A est restriction de B, notée  $\text{restr}(B)$  par

$$- A \subset B$$

$$- \forall R \in B-A \exists R' \in A \text{ tel que } R \subset R'$$

Toute relation de B-A est dite être inutile.

La proposition 5

$$A = \text{restr}(B) \implies \bigvee_{R' \in A} R' = \bigvee_{R \in B} R$$

est évidente.

Appliquons ces résultats aux ensembles  $D_p(X_i)$  de l'algorithme A1 :

Définissons des suites d'ensembles  $D'_p(X_i)$  par :

$$D'_0(X_i) = \phi$$

$$D'_p(X_i) = \text{restr}(\bigvee_j \text{pr}(\phi_j(D(r_j))) \vee \dots \vee \phi_n(D(r_n)) \vee DE(i,j))^+$$

$$\text{pour } 1 \leq j \leq p_i \text{ et } \forall k \ 1 \leq k \leq n \ D(r_k) \in D'_{p-1}(X_{ijk})$$

Alors il est évident de montrer, par récurrence sur p, la proposition 6 :

$$D'_p(X_i) = \text{restr}(D_p(X_i)).$$

On peut, bien sûr, également appliquer cette notion de restriction aux ensembles  $D(X_i)$  de l'algorithme A3, ce qui permet de ne conserver que les relations utiles à la détection de circularité.

On en déduit le théorème 7 :

*Théorème 7* : on conserve la puissance de détection de circularité de l'algorithme A3 en éliminant des ensembles  $D(X_i)$  les relations inutiles.

#### 10.8.2. Première itération.

On a, dans l'algorithme A1, à l'initialisation des itérations,

$$D_0(X_i) = \phi \quad \forall X_i \in N.$$

On en déduit qu'à la première itération, on a

$$D_1(X_i) = \bigvee_j \text{pr}(DE(i,j))^+ \quad \text{pour } 1 \leq j \leq p_i$$

On peut donc gagner une itération en initialisant, dans l'algorithme A3,  $D(X_i)$  par

$$D(X_i) = \bigvee_j \text{pr}(DE(i,j))^+ \quad \text{pour } 1 \leq j \leq p_i$$

#### 10.8.3. Algorithme A4.

Cet algorithme s'obtient par application des deux résultats précédents, à partir de l'algorithme A3.

Nous notons  $N_k$  le nombre d'éléments de l'ensemble  $D(X_{ijk})$  initialisation :

pour  $i := 0$  pas 1 jusqu'à  $m$  faire

$$. D(X_i) = \text{pr}(DE(i,1))^+$$

. pour  $j := 2$  pas 1 jusqu'à  $p_i$  faire

$$. \text{construire } R_1 = \text{pr}(DE(i,j))^+$$

. tant que  $\exists R_2 \in D(X_i) : R_2 \subset R_1 \Rightarrow \text{exclure } R_2$ .  
puis inclure  $R_1$

sinon.  $\exists R_2 \in D(X_i) : R_1 \subset R_2 \Rightarrow \text{ne pas inclure } R_1$

sinon. inclure  $R_1$

```

jusqua convergence faire
pour i := 0 pas 1 jusqu'a m faire
  pour j := 1 pas 1 jusqu'a pi faire
    pour k1 := 1 pas 1 jusqu'a N1 faire
      ...
      pour kn := 1 pas 1 jusqu'a Nn faire
        . construire R1 = pr(D(Xij1) [k1] ∨ ... ∨ D(Xijn) [kn]
          ∨ DE(i,j))+
        . tant que ∃ R2 ∈ D(Xi) : R2 ⊂ R1 ⇒ exclure R2
          puis inclure R1
        sinon. ∃ R2 ∈ D(Xi) : R1 ⊂ R2 ⇒ ne pas inclure R1
        sinon. inclure R1

```

#### 10.8.4. Exemple.

Suivons le déroulement de l'algorithme A4 sur le premier exemple du § 10.4.2.1.

$$\begin{aligned}
 D(A) &= \{a \ b \ c \ d\} \\
 D(B) &= \{a \rightarrow b \ c \ d, \ a \ b \ c \rightarrow d\} \\
 D(A) &= \{a \rightarrow b \rightarrow c \rightarrow d, \ a \leftarrow b \leftarrow c \leftarrow d\} \\
 D(B) &= \{a \rightarrow b \ c \ d, \ a \ b \ c \rightarrow d\}
 \end{aligned}$$

La convergence est atteinte.

#### 10.8.5. Applicabilité de l'algorithme A4.

Les améliorations que nous venons d'apporter font que l'algorithme A4 est utilisable pratiquement alors que l'algorithme A3, sans élimination des relations inutiles, conduit très vite à une occupation mémoire déraisonnable pour des langages non triviaux. Le facteur de réduction de l'espace mémoire est de l'ordre de 10, en moyenne. Il faut également noter que le nombre de fois où la fermeture transitive est calculée est considérablement réduit, au détriment de tests d'appartenance à des ensembles de relations.

L'algorithme A4 est, pratiquement, le seul applicable de tous nos algorithmes. Les deux algorithmes A2 sont utiles et simples mais, malheureusement, insuffisants.

## 11. SYSTEME DELTA

Le chapitre présente l'état actuel du système DELTA (DEfinition de Langages et de Traducteurs par Attributs sémantiques), en tant que méta-compilateur. Les principes de définition du système sont donnés mais sans descendre jusqu'aux détails techniques de réalisation.

### 11.1. Organisation générale d'un métacompilateur.

Un système de métacompilation a pour but d'automatiser l'écriture de traducteurs. Aussi est-il formé d'un ensemble de composants, chargés de traiter chacun des aspects d'un processus de traduction. Dans la majorité des cas, on trouve :

- un constructeur d'analyseur lexicographique qui accepte, en entrée, la description des atomes du langage (mots clés, identificateurs, séparateurs, délimiteurs, etc) et produit, en sortie, un analyseur lexicographique. On peut, par exemple, consulter les références [42] et [43].
  - un constructeur syntaxique qui, traitant la description syntaxique du langage écrite dans un métalangage du type BNF, produit un analyseur syntaxique. Dans la plupart des cas, le constructeur est écrit pour n'accepter qu'un certain type de grammaires ; dans d'autres cas, plus rares, il essaie de modifier et de transformer la grammaire donnée pour l'adapter au type de grammaire prévue. La bibliographie sur ce sujet est innombrable. Citons l'école d'Eté de Munich 1974 qui aborde largement ce domaine ([44], [45]) :
  - le traitement de la sémantique peut être incorporé à la description syntaxique, comme, par exemple, dans le système CDL (Compiler Description Language) qui utilise la notion de grammaires affixes, ou associé à chaque règle de grammaire, comme dans la méthode de Wirth-Weber. La sémantique est donc décrite de façon algorithmique, l'évaluation étant, le plus souvent, parallèle à l'analyse syntaxique. Cette structure générale d'un métacompilateur n'est, bien sûr, pas immuable, mais elle se retrouve très largement dans la plupart des systèmes de compilation de compilateurs.
- Les objectifs de DELTA sont ceux d'un métacompilateur. Aussi, son organisation est-elle très voisine de celle que nous venons de décrire. On peut distinguer trois phases de traitement dans DELTA.

- La construction syntaxique qui bâtit un analyseur syntaxique à partir de la définition syntaxique ; la construction sémantique qui traite la définition sémantique.

Notons que cette phase s'applique sur la définition du langage, alors que les deux phases suivantes s'appliquent à un programme de ce langage.

- L'analyse syntaxique d'un programme, où le système d'équations est construit.
- La résolution de ce système d'équations, ou évaluation sémantique. Les paragraphes qui suivent présentent le système pour chacune de ces phases.

## 11.2. Traitement à la construction.

### 11.2.1. *Constructeurs syntaxiques de DELTA.*

La partie "description syntaxique" du langage est traitée par un constructeur lexicographique et par des constructeurs syntaxiques qui sont au nombre de trois ; ils acceptent des grammaires :

- . SLR (1) , sous ensemble simple des grammaires LR(1),
- . LALR (1) , sous ensemble plus large des grammaires LR(1),
- . LR (1)

Pour les définitions précises de ces grammaires, voir [47]

Ce choix de grammaires LR(1), vient du fait qu'on a la preuve théorique que les grammaires LR(k) constituent la plus large classe des grammaires hors-contexte, que l'on puisse analyser de façon déterministe de gauche à droite. Le choix de  $k = 1$  vient de considérations pratiques évidentes.

Les raisons des deux sous ensembles des grammaires LR(1) viennent aussi de considérations pratiques. La complexité en temps et en place de la construction peut devenir très grande pour des grammaires LR(1) non triviales et, dans bon nombre de cas, cette complexité est superflue. Pour des raisons d'homogénéité de construction, il est apparu très naturel de rechercher des sous-ensembles des grammaires LR(1) plutôt que de changer de méthodes. Cette approche offre, de plus, l'avantage de n'avoir qu'un seul analyseur quel que soit le constructeur qui a été utilisé.

La raison d'être des deux sous-ensembles des LR(1) est que, si les LR(1) sont souvent trop puissantes, les SLR(1) ne le sont pas assez, d'où cette classe intermédiaire, les LALR(1), de complexité et de puissance acceptables.

La démarche que nous préconisons est d'utiliser les trois constructeurs, par puissance croissante, et de s'arrêter à celui qui convient à la grammaire proposée. L'expérience nous a montré que le SLR(1) suffit souvent, le LALR(1) presque toujours.

Nous avons rencontré peu d'exemples qui ne soient pas LALR(1) mais qui soient LR(1), hormis des exemples d'écoles, bien sûr.

Nous n'entrerons pas plus en détail dans ces aspects syntaxiques. Le lecteur intéressé trouvera, dans un rapport du projet DELTA, [48], les méthodes de construction utilisées, où un gros effort a été fait sur l'optimisation des analyseurs syntaxiques produits.

### 11.2.2. Constructeur sémantique de DELTA.

#### 11.2.2.1. Principes d'implantation.

La différence fondamentale entre les métacompilateurs classiques et DELTA tient à la définition et à l'évaluation sémantique des programmes.

Le constructeur sémantique analyse les définitions sémantiques associées à chaque règle et construit un ensemble d'informations pour les phases ultérieures.

Quels sont ces informations ?

Tout d'abord, pour chaque symbole, il est nécessaire de connaître les règles où il est défini. Puis, pour chaque règle, il faut connaître les symboles qui y apparaissent, les définitions sémantiques associées que nous appellerons modèles sémantiques. Chaque modèle décrit les attributs mis en oeuvre ainsi que le calcul de l'attribut sujet du modèle.

Ces constatations conduisent naturellement à la structure de données suivante, que nous illustrerons à l'aide de l'exemple du § 9.4., en ne s'intéressant qu'à la règle :

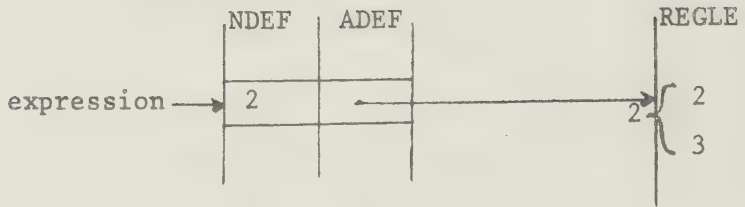
```

<expression> := <expression> <pm> <opérande>
g(expression') := vrai
g(opérande) := faux
r(expression) := si r(expression') = r(opérande)
                  alors r(expression') + 1
                  sinon max(r(expression'), r(opérande))

```

Le symbole <expression> donne une entrée dans deux tableaux parallèles qui indiquent le nombre de définitions du symbole et leurs numéros dans un tableau des règles.

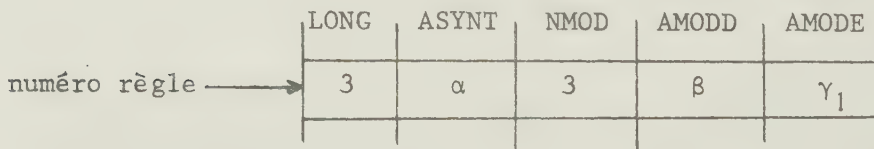
On a donc



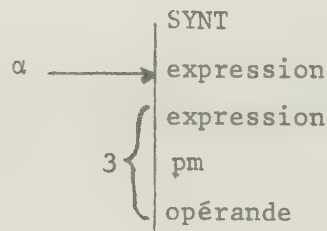
<expression> a 2 définitions (nous ne considérerons ici que la première). L'accès indirect aux règles a pour but de faciliter les modifications d'une grammaire.

Le tableau des règles contient des numéros de règles (ici c'est la 2ème production) qui servent d'entrée dans cinq tableaux parallèles qui donnent

- la longueur de la production
- l'adresse de la description syntaxique
- le nombre de définitions sémantiques
- l'adresse de description du premier modèle de définition sémantique
- l'adresse de description du premier modèle d'évaluation sémantique

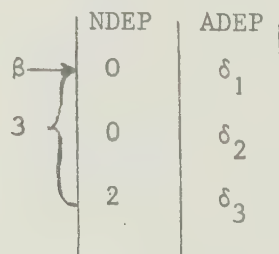


la description syntaxique comprend les symboles de la règle :



Les modèles de définition sémantique comprennent :

- le nombre de dépendance
- l'adresse des relations de dépendance



Les relations de dépendance sont les relations DE codifiées :

|                        | DEP |
|------------------------|-----|
| $\delta_1 \rightarrow$ | g 1 |
| $\delta_2 \rightarrow$ | g 3 |
| $\delta_3 \rightarrow$ | r 0 |
|                        | r 1 |
|                        | r 3 |

Les modèles d'évaluation sémantique donnent le texte codifié des définitions :

|                                                        | TAILLE | TEXTE                                                                              |
|--------------------------------------------------------|--------|------------------------------------------------------------------------------------|
| $\gamma \rightarrow$                                   | $t_1$  | $g_1 := \text{vrai}$                                                               |
| $\left. \begin{array}{c} 3 \\ \} \end{array} \right\}$ | $t_2$  | $g_3 := \text{faux}$                                                               |
|                                                        | $t_3$  | $r_0 := \text{si } r_1 = r_3 \text{ alors } r_1 + 1 \text{ sinon } \max(r_1, r_3)$ |

### 11.2.2.2. Algorithmes de tests de cohérence.

Les tests de cohérence ne sont pas systématiquement faits sur une définition sémantique car ils sont extrêmement coûteux en place mémoire et temps. L'utilisateur du système les demande ou non, mais le système considère, pour les phases ultérieures, qu'ils ont été faits.

#### a) Unicité de définition.

L'algorithme du paragraphe 10.3. est appliqué. Il traite successivement les attributs synthétisés puis hérités. A chaque symbole est associée une liste de ses attributs sous la forme :

|                          | NATS | AATS       | NATH | AATH       |
|--------------------------|------|------------|------|------------|
| expression $\rightarrow$ | 1    | $\alpha_s$ | 1    | $\alpha_h$ |

|                        | ATS | ATH |
|------------------------|-----|-----|
| $\alpha_s \rightarrow$ | r   |     |
| $\alpha_h \rightarrow$ |     | g   |

L'algorithme parcourt la structure de données jusqu'à atteindre les relations de dépendances qui donnent, pour chaque règle, les attributs définis.

Soulignons l'importance de ces tests. Si la multi-définition est une erreur naïve que l'utilisateur pourrait trouver sans difficulté, l'indéfinition d'attributs hérités est fréquente. En effet, il n'est pas aisé de retrouver sur une description importante, toutes les occurrences d'un symbole en partie droite d'une règle et de vérifier que ses attributs hérités sont bien tous définis.

Citons, par exemple, l'oubli fréquent de la définition de  $g(\text{expression})$  dans la définition de l'axiome, pour l'algorithme de Sethi-Ullman. Or cette définition est nécessaire pour une expression limitée à un seul opérande.

#### b). Tests de circularité.

C'est l'algorithme du paragraphe 10.8.3. qui est implanté. Le problème essentiel de cet algorithme est le calcul de l'union des dépendances et de la fermeture transitive, ainsi que la représentation mémoire des dépendances. Ces ensembles de dépendance  $D$  contiennent les dépendances superficielles : ils sont mémorisés sous forme de matrices carrées booléennes

$$D(x,y) \iff (x,0) D (y,0) \text{ noté } x_0 D y_0$$

L'union des dépendances est réalisée dans une matrice carrée booléenne  $U$  adressée de la façon suivante :

Pour chaque règle

$$x_{ij0} \longrightarrow x_{ij1} \dots x_{ijn_{ij}}$$

on considère que chaque attribut peut apparaître  $n_{ij} + 1$  fois ; s'il y a  $n$  attribut possibles, on construit une matrice  $U$  d'ordre  $n \times (n_{ij} + 1)$  qui a la signification suivante :

$$U(a + kn, b + ln) \iff (a,k) D (b,l) \text{ noté } a_k D b_l$$

Pour chaque règle, une matrice  $DE$  est construite selon le même principe que  $U$ .

Reprenons l'exemple du paragraphe 10.5.3.

$A \longrightarrow B$

|       |    |       |
|-------|----|-------|
| $a_0$ | DE | $d_0$ |
| $c_0$ | DE | $b_0$ |
| $a_1$ | DE | $a_0$ |
| $b_0$ | DE | $b_1$ |
| $c_1$ | DE | $c_0$ |
| $d_0$ | DE | $d_1$ |

DE =

|       |       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|
|       | $a_0$ | $b_0$ | $c_0$ | $d_0$ | $a_1$ | $b_1$ | $c_1$ | $d_1$ |
| $a_0$ |       |       |       | 1     |       |       |       |       |
| $b_0$ |       |       |       |       |       | 1     |       |       |
| $c_0$ |       | 1     |       |       |       |       |       |       |
| $d_0$ |       |       |       |       |       |       |       | 1     |
| $a_1$ | 1     |       |       |       |       |       |       |       |
| $b_1$ |       |       |       |       |       |       |       |       |
| $c_1$ |       |       | 1     |       |       |       |       |       |
| $d_1$ |       |       |       |       |       |       |       |       |

$B \longrightarrow x$

|       |    |       |
|-------|----|-------|
| $b_0$ | DE | $a_0$ |
|-------|----|-------|

DE =

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
|       | $a_0$ | $b_0$ | $c_0$ | $d_0$ |
| $a_0$ |       |       |       |       |
| $b_0$ | 1     |       |       |       |
| $c_0$ |       |       |       |       |
| $d_0$ |       |       |       |       |

$B \longrightarrow y$

|       |    |       |
|-------|----|-------|
| $d_0$ | DE | $c_0$ |
|-------|----|-------|

DE =

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
|       | $a_0$ | $b_0$ | $c_0$ | $d_0$ |
| $a_0$ |       |       |       |       |
| $b_0$ |       |       |       |       |
| $c_0$ |       |       |       |       |
| $d_0$ |       |       | 1     |       |

Pour indiquer clairement l'opération d'union et de fermeture transitive, nous supposons que la première itération a été réalisée. On a donc :

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
|       | $a_0$ | $b_0$ | $c_0$ | $d_0$ |
| $a_0$ |       |       |       | 1     |
| $b_0$ |       |       |       |       |
| $c_0$ |       | 1     |       |       |
| $d_0$ |       |       |       |       |

$$D(B) = \begin{array}{c} a_0 \ b_0 \ c_0 \ d_0 \\ \begin{array}{|c|c|c|c|} \hline a_0 & & & \\ \hline b_0 & 1 & & \\ \hline c_0 & & & \\ \hline d_0 & & & \\ \hline \end{array} \end{array}, \begin{array}{c} a_0 \ b_0 \ c_0 \ d_0 \\ \begin{array}{|c|c|c|c|} \hline a_0 & & & \\ \hline b_0 & & & \\ \hline c_0 & & & \\ \hline d_0 & & 1 & \\ \hline \end{array} \end{array}$$

Explicitons le calcul de  $(D_1(B) \vee DE(1))^+$

On part d'une matrice  $U$  égale à  $DE(1)$

$$U = \begin{array}{c} a_0 \ b_0 \ c_0 \ d_0 \ a_1 \ b_1 \ c_1 \ d_1 \\ \begin{array}{|c|c|c|c|c|c|c|c|} \hline a_0 & & & 1 & & & & \\ \hline b_0 & & & & & 1 & & \\ \hline c_0 & & 1 & & & & & \\ \hline d_0 & & & & & & & 1 \\ \hline a_1 & 1 & & & & & & \\ \hline b_1 & & & & & & & \\ \hline c_1 & & & 1 & & & & \\ \hline d_1 & & & & & & & \\ \hline \end{array} \end{array}$$

avec laquelle on fait l'union de  $D_1(B)$  transformée. En effet  $D_1(B)$  suppose que  $B$  a pour indice 0 alors qu'il doit avoir l'indice 1 pour la règle  $A \rightarrow B$ . On transforme  $b_0 DE a_0$  en  $b_1 DE a_1$ . Ce qui donne la matrice  $U$  finale, dont on calcule la fermeture transitive.

$$U =$$

|       | $a_0$ | $b_0$ | $c_0$ | $d_0$ | $a_1$ | $b_1$ | $c_1$ | $d_1$ |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| $a_0$ |       |       |       | 1     |       |       |       |       |
| $b_0$ |       |       |       |       |       |       | 1     |       |
| $c_0$ |       | 1     |       |       |       |       |       |       |
| $d_0$ |       |       |       |       |       |       |       | 1     |
| $a_1$ | 1     |       |       |       |       |       |       |       |
| $b_1$ |       |       |       |       |       | ①     |       |       |
| $c_1$ |       |       | 1     |       |       |       |       |       |
| $d_1$ |       |       |       |       |       |       |       |       |

$$U^+ =$$

|       | $a_0$ | $b_0$ | $c_0$ | $d_0$ | $a_1$ | $b_1$ | $c_1$ | $d_1$ |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| $a_0$ |       |       |       | 1     |       |       |       | 1     |
| $b_0$ | 1     |       |       | 1     | 1     | 1     |       | 1     |
| $c_0$ | 1     | 1     |       | 1     | 1     | 1     |       | 1     |
| $d_0$ |       |       |       |       |       |       |       | 1     |
| $a_1$ | 1     |       |       | 1     |       |       |       | 1     |
| $b_1$ | 1     |       |       | 1     | 1     |       |       | 1     |
| $c_1$ | 1     | 1     | 1     | 1     | 1     | 1     |       | 1     |
| $d_1$ |       |       |       |       |       |       |       |       |

On ne retient de la matrice  $U^+$  que la partie correspondant à l'indice 0, qui constitue les dépendances superficielles recherchées.

Essayons d'obtenir une estimation de la place en mémoire demandée par l'algorithme précédent pour une grammaire de 200 symboles non terminaux, 20 attributs et 5 symboles en moyenne par règle.

Dans l'implantation effective, les matrices DE ne sont pas construites car on a l'expression de DE sous forme de liste dans le tableau DEP ; nous ne les avons introduites ici que par souci de clarté.

La matrice  $U$  est construite à chaque calcul : elle occupe  $(20 \times 5)^2 = 10000$  éléments. Cette place peut être réutilisée pour le calcul de la fermeture transitive.

Chaque matrice  $E$  occupe  $20^2 = 400$  éléments. Avec les améliorations signalées au paragraphe 10.5.6, on a rarement plus de 2 matrices par symbole, le plus souvent une seule matrice suffit, soit  $400 \times 200 = 80000$  éléments, alors que l'algorithme initial nécessite un nombre de matrices qui atteint très vite 10 ; ce qui explique qu'il soit très vite inapplicable.

L'implantation que nous venons de décrire est donc raisonnable en place mémoire.

Ces tests de cohérence sont fondamentaux pour la méthode, dans la mesure où ils apportent la preuve que toute évaluation sémantique aboutit. On pourrait se contenter, comme dans le système FOLDS [31] de tests systématiques d'incohérence au cours de l'évaluation. Mais les avantages de ces tests à la construction du traducteur sont évidents. Pour paraphraser Dijkstra, des tests à l'évaluation n'apporteront jamais la preuve de la cohérence de la

définition, ils n'apportent, parfois, que des preuves d'incohérence.

### 11.3. Traitement à l'analyse syntaxique.

Quel que soit le constructeur syntaxique utilisé, l'analyse syntaxique est du type ascendant déterministe gauche-droite.

#### 11.3.1. *Décoration des ramifications.*

D'après ce que nous avons vu au paragraphe 9.5., la décoration, ou numérotation des noeuds de l'arbre syntaxique, a lieu au cours de l'analyse syntaxique ascendante. Pour cela, une pile sémantique parallèle à la pile d'analyse va contenir à tout instant le numéro attribué par le système à chaque noeud. Toutes les informations nécessaires à la phase d'évaluation sémantique sont établies et mémorisées. Nous allons supposer, tout d'abord que le système d'équations est construit.

Prenons encore la règle de l'algorithme de Sethi Ullman que nous avons décrite à la phase de construction et étudions une réduction de cette règle.

Avant la réduction nous avons les piles syntaxiques et sémantiques suivantes :

|                         |   |                        |
|-------------------------|---|------------------------|
| opérande                | 9 | } LONG(règle n° 2) = 3 |
| pm                      | 3 |                        |
| expression              | 2 |                        |
| ⋮                       |   |                        |
| syntaxe      décoration |   |                        |

La réduction va entraîner le remplacement des 3 éléments au sommet des piles par expression, décoré par le nombre 10, par exemple.

#### 11.3.2. *Equations sémantiques.*

Par consultation du tableau NMOD, le système sait qu'il a trois modèles à traiter. Il doit remplacer, dans les relations de dépendance :

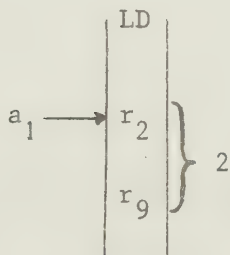
1    par    2  
3    par    9  
0    par    10,

informations extraites de la pile sémantique.

Les équations sont alors obtenues et associées aux relations de dépendance élémentaire

| Attribut                 | Nombre de<br>dépendance | dépendance | Equation                                                                              |
|--------------------------|-------------------------|------------|---------------------------------------------------------------------------------------|
| $g_2 \longrightarrow$    | 0                       | /          | $g_2 := \text{vrai}$                                                                  |
| $g_9 \longrightarrow$    | 0                       | /          | $g_9 := \text{faux}$                                                                  |
| $r_{10} \longrightarrow$ | 2                       | $a_1$      | $r_{10} := \text{si } r_2 = r_9 \text{ alors } r_2 + 1 \text{ sinon max } (r_2, r_9)$ |

$a_1$  est une entrée dans une liste de dépendance



Mais la mémorisation des équations est extrêmement coûteuse en place et nous avons préféré conserver toutes les informations pour les reconstituer lors de l'évaluation sémantique.

La représentation adoptée est la suivante

| Attribut                 | Nombre de<br>dépendance | adresse liste | adresse modèle |
|--------------------------|-------------------------|---------------|----------------|
| $g_2 \longrightarrow$    | 0                       | $a_1$         | $\gamma_1$     |
| $g_9 \longrightarrow$    | 0                       | $a_2$         | $\gamma_2$     |
| $r_{10} \longrightarrow$ | 2                       | $a_3$         | $\gamma_3$     |

La liste de dépendance est alors

|                   | LD       |
|-------------------|----------|
| $a_1 \rightarrow$ | $g_2$    |
| $a_2 \rightarrow$ | $g_9$    |
| $a_3 \rightarrow$ | $r_{10}$ |
|                   | $r_2$    |
|                   | $r_9$    |

Les modèles d'évaluation sémantique sont alors mémorisés sous une forme légèrement différente de celle présentée auparavant

|                            | TAILLE | TEXTE                                                                              |
|----------------------------|--------|------------------------------------------------------------------------------------|
| $\gamma_1 \longrightarrow$ | $t_1$  | $g_0 := \text{vrai}$                                                               |
| $\gamma_2 \longrightarrow$ | $t_2$  | $g_0 := \text{faux}$                                                               |
| $\gamma_3 \longrightarrow$ | $t_3$  | $r_0 := \text{si } r_1 = r_2 \text{ alors } r_1 + 1 \text{ sinon } \max(r_1, r_2)$ |

Dans les textes, les indices des inconnues renvoient sur la liste de dépendance par l'intermédiaire de l'adresse de liste.

#### 11.4. Généralités sur l'évaluation sémantique.

##### 11.4.1. *Introduction.*

Ce paragraphe analyse les différentes méthodes d'évaluation sémantique. Il y en a fondamentalement deux : la première part du système d'équations et essaie de le résoudre en ne tenant pas compte du graphe de dépendance ; la deuxième exploite systématiquement le graphe de dépendance pour résoudre le système. Cette deuxième approche nous a semblé plus séduisante et c'est elle qui est utilisée dans DELTA.

##### 11.4.2. *Evaluation sémantique par processus coopérants.*

Dans le système de Fang, à l'université de Stanford, la solution adoptée est de considérer chaque inconnue du système comme un processus, au sens de processus asynchrone. L'état d'un tel processus peut être :

- terminé : l'équation correspondante est résolue, la valeur de l'inconnue est disponible,
- actif : l'équation est en cours de résolution,
- bloqué : la résolution n'a pu se poursuivre car elle nécessite un autre processus qui est actif ou bloqué.

L'activation des processus est faite aléatoirement, un processus reste actif tant que, dans son équation, les inconnues rencontrées correspondent à des processus terminés dont les valeurs sont donc disponibles ; il se termine lorsque l'équation est complètement résolue ; il se bloque dès la rencontre d'une inconnue dont le processus est actif ou bloqué, ce blocage étant levé avec passage à nouveau, à l'état actif dès que ce processus est terminé.

Une question que les spécialistes de systèmes ont déjà dû poser est : a-t-on la preuve qu'une telle évaluation se termine quel que soit l'ordre d'activation des processus ? La réponse est, bien sûr : oui, dans tous les cas, si la définition sémantique est cohérente. En particulier, le problème de l'étreinte fatale est, a priori, résolu si la définition ne contient pas de circularité ; à tout processus activable correspond une équation s'il n'y a pas d'indéfinition ; à tout processus correspond une seule équation s'il n'y a pas de multidéfinition. On voit, ici, l'importance de la notion de cohérence. On voit aussi que l'activation de processus est tout à fait imaginable et même souhaitable sur une structure de machine multi-processeur. Nous reviendrons sur ce point au chapitre 12.

La critique principale que l'on peut faire à ce type d'évaluation est le côté aléatoire de l'activation de processus. Il néglige complètement les dépendances entre inconnues, qui sont exprimées dans les équations. Mais quelle meilleure preuve du caractère déclaratif des définitions par attributs peut-on apporter que de pouvoir affirmer que la résolution d'un système est la même quel que soit l'ordre de calcul des inconnues ? Ce souci de demeurer déclaratif jusqu'au bout explique ce choix qui est fait dans FOLDS. La démonstration est éclatante par rapport aux autres méthodes traductives.

Nous avons signalé une critique générale à la méthode par attributs, le nombre élevé d'inconnues mises en oeuvre. Or, la notion de processus de FOLDS est attachée à chaque attribut de chaque noeud de l'arbre. Et on voit donc mal comment espérer réduire ce nombre.

### 11.4.3. *Évaluation sémantique par recherche d'un ordre de résolution.*

#### 11.4.3.1. Introduction, définitions.

Supposons que ce sont les relations de dépendance construites avec le système, qui sont exploitées. Nous visualiserons toujours ces relations sous forme de graphes de dépendance, pour illustrer l'évaluation sémantique. Nous appellerons GD un tel graphe.

A l'aide des relations de dépendance, il nous faut parcourir le graphe pour évaluer les inconnues situées à ses noeuds.

Définition. On appelle parcours ascendant du graphe, le parcours effectué à l'aide de la relation DE. Le parcours est descendant lorsqu'on utilise la relation inverse  $DE^{-1}$ .

- on note  $DE(a)$  l'ensemble, défini pour toute inconnue  $a$ , des inconnues impliquant  $a$  :

$$DE(a) = \{b \mid a \text{ DE } b\}$$

- on note  $DE^{-1}(a)$  l'ensemble, défini pour toute inconnue  $a$ , des inconnues impliquées par  $a$  :

$$DE^{-1}(a) = \{b \mid b \text{ DE } a\}$$

- on appelle IE, pour Inconnues d'Entrée, l'ensemble défini par

$$IE = \{a \mid DE(a) = \emptyset\}$$

- on appelle IS, pour Inconnues de Sortie, l'ensemble défini par

$$IS = \{a \mid DE^{-1}(a) = \emptyset\}$$

Nous allons maintenant étudier les parcours ascendants et descendants. On doit noter que le but de ces parcours est, bien sûr, l'évaluation des inconnues situées aux noeuds du graphe.

Par évaluation, on peut comprendre deux choses :

- l'évaluation proprement dite, c'est-à-dire le calcul effectif de la valeur de l'inconnue avec conservation au noeud de cette valeur,
- en considérant la définition d'un attribut comme une macro opération, l'émission du texte de cette définition paramétré par les noeuds du graphe sous forme d'une équation, pour constituer un programme d'évaluation.

La première approche permet d'effectuer l'évaluation dans le graphe, la seconde d'émettre les équations dans un ordre où elles sont résolubles.

Actuellement, le système DELTA ne permet que la deuxième approche mais la première est prévue à court terme.

#### 11.4.3.2. Parcours ascendant du graphe.

Ce type de parcours correspond à un ordre de visite depuis les noeuds d'entrée vers les noeuds de sortie.

L'algorithme utilisé est le tri topologique ([35] page 258),

.  $\exists a : DE(a) = \phi \implies \left\{ \begin{array}{l} \text{évaluer } (a) \\ \forall b \in DE^{-1}(a) \text{ faire } DE(b) = DE(b) - a \\ \text{éliminer } (a) \text{ du graphe} \\ \text{recommencer} \end{array} \right.$

. la visite est terminée.

Cet algorithme est extrêmement simple : tout noeud n'ayant pas d'impliquant est visité et est enlevé de l'ensemble des impliquants des noeuds qu'il implique, puis est éliminé du graphe.

On doit remarquer que cet algorithme est implantable sur des calculateurs à structure parallèle\*, tous les candidats à visite pouvant l'être en parallèle.

#### 11.4.3.3. Parcours descendant du graphe.

Cet ordre de parcours correspond au parcours en profondeur [36]. L'ordre de visite est depuis les noeuds de sortie vers les noeuds d'entrée.

Les noeuds visités en premier correspondent aux attributs évaluable en dernier. L'ordre de visite correspond donc à l'ordre inverse d'évaluation.

.  $\exists a : DE^{-1}(a) = \phi \implies \left\{ \begin{array}{l} \text{visiter } (a) \\ \forall b \in DE(a) \text{ faire } DE^{-1}(b) = DE^{-1}(b) \\ \text{éliminer } (a) \\ \text{recommencer} \end{array} \right.$

. la visite est terminée et l'évaluation se fait dans l'ordre inverse de visite des noeuds.

Cet algorithme est également implantable en parallèle pour la phase de visite ainsi que pour la phase d'évaluation mais nécessite la mémorisation des

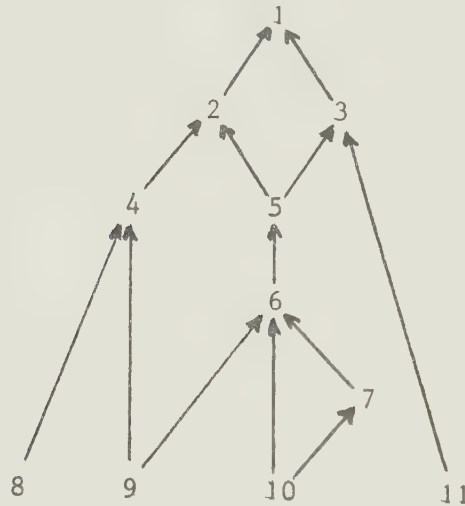
---

\* ou multiprocesseurs.

noeuds visités pour pallier à l'élimination faite pendant la visite.

#### 11.4.3.4. Exemple.

Considérons le graphe de dépendance ci-joint



. algorithme ascendant

|   |   |    |    |
|---|---|----|----|
| 8 | 9 | 10 | 11 |
| 4 | 7 |    |    |
|   | 6 |    |    |
|   | 5 |    |    |
| 2 | 3 |    |    |
| 1 |   |    |    |

. algorithme descendant

|    |   |    |
|----|---|----|
| 1  |   |    |
| 2  | 3 |    |
| 4  | 5 | 11 |
| 6  | 8 |    |
| 7  | 9 |    |
| 10 |   |    |

#### 11.4.3.5. Commentaires.

Ces algorithmes élémentaires permettent de placer en parallèle des noeuds de même distance par rapport aux noeuds d'entrée et aux noeuds de sortie. Cette distance est supposée égale à 1 ici. En fait, elle correspond au temps d'évaluation d'une inconnue placée à un noeud. Nous avons supposé ce temps égal pour toutes les inconnues. Ce n'est bien sûr, pas vrai dans la réalité

où le temps d'évaluation d'une inconnue n'est, en général, pas indépendant de son occurrence dans le graphe.

L'évaluation en parallèle nécessite, bien sûr, une structure de machine adaptée. Nous y reviendrons au chapitre 12 en reliant ce problème à celui de langages parallèles et de machine parallèle.

Mais, dans l'immédiat, une telle structure de machine n'est pas très répandue et il nous faut nous intéresser aux calculateurs monoprocesseurs, ce qui est le cas jusqu'à la fin de ce chapitre.

#### 11.4.4. *Évaluation sémantique en alternance avec l'analyse syntaxique gauche - droite.*

##### 11.4.4.1. Introduction

Jusqu'à maintenant, nous avons considéré que l'évaluation sémantique était complètement dissociée de l'analyse syntaxique. Mais il est très naturel de penser à les faire alterner pour effectuer la résolution du système associé à une règle dès que l'analyse syntaxique a reconnu cette règle. Réaliser l'évaluation sémantique au fur et à mesure de l'analyse syntaxique est la stratégie adoptée dans tous les traducteurs dirigés par la syntaxe.

On doit, tout de suite, remarquer qu'une définition sémantique demandant plusieurs passes sur l'arbre de dérivation exclut, a priori, une évaluation en alternance avec une analyse gauche-droite. On peut malgré tout penser qu'en apportant des restrictions aux fonctions de définition des attributs, on pourrait réaliser une telle évaluation. Intuitivement, on peut penser que les dépendances élémentaires entre attributs doivent respecter l'ordre dans lequel l'arbre syntaxique est construit au cours de l'analyse.

Cette remarque est confirmée par deux études faites sur ce sujet ([37] et [38]).

A la règle

$$X_{ij0} \longrightarrow X_{ij1} \dots X_{ijn_{ij}}$$

nous avons associé un système de définition  $SE(i,j)$  qui définit tous les attributs des symboles de la production.

Nous allons donner des restrictions sur les ensembles de dépendance des attributs définis, fonctions du type d'analyse syntaxique.

#### 11.4.4.2. Evaluation sémantique en alternance avec une analyse syntaxique descendante déterministe gauche droite.

$$R1 : DE(a_0) \subset \{b_0 \mid b \in H(X_{ij0})\} \cup \{c_1 \mid 1 \leq l \leq n_{ij}, c \in S(X_{ij1})\}$$

$$R2 : DE(a_k) \subset \{b_0 \mid b \in H(X_{ij0})\} \cup \{c_1 \mid 1 \leq l \leq k-1, c \in S(X_{ij1})\}$$

$$1 \leq k \leq n_{ij}$$

Ces contraintes ne font que confirmer nos remarques préalables.

Ces restrictions sont, de façon informelle, pour une règle :

- R1 : les attributs synthétisés du symbole non terminal en partie gauche ne dépendent que des attributs hérités de ce même symbole et des attributs synthétisés des symboles en partie droite.
- R2 : les attributs hérités d'un symbole en partie droite ne dépendent que des attributs hérités du symbole en partie gauche et des attributs synthétisés des symboles, en partie droite, situés à sa gauche.

L'évaluation se déroule alors ainsi :

- . tous les attributs hérités du symbole en partie gauche sont évalués
- . ils sont communiqués aux symboles en partie droite
- . ces symboles synthétisent leurs attributs qui sont communiqués aux symboles plus à droite dans la règle
- . les attributs synthétisés du symbole en partie gauche sont évalués et communiqués au point d'appel.

Une remarque doit être faite : les attributs synthétisés du symbole en partie gauche ne peuvent pas dépendre d'autres attributs synthétisés du même symbole. Cette restriction, un peu surprenante, s'explique par le fait qu'on ne veut imposer aucun ordre d'évaluation de ces attributs synthétisés, alors qu'autoriser des interdépendances impliquerait un ordonnancement avant leur évaluation.

Les implanteurs connaissent bien les restrictions précédentes, par lesquelles ils doivent exprimer les processus de traduction. Les actions sémantiques placées avant un symbole dans une production, correspondent à l'héritage d'informations, via l'environnement global, vers ce symbole ; placées après un symbole, elles permettent de synthétiser, via l'environnement global, des informations depuis ce symbole. Quant aux actions en fin de production, elles correspondent à la synthèse globale de la production, toujours

par l'intermédiaire d'un environnement global.

#### 11.4.4.3. Evaluation sémantique en alternance avec une analyse ascendante déterministe gauche droite.

$$R3 : DE(a_0) \subset \{b_l \mid 1 \leq l \leq n_{ij}, b \in S(X_{ijl})\}$$

Cette restriction est plus sévère que précédemment : seuls les attributs synthétisés sont autorisés.

- . Les attributs synthétisés de tous les symboles en partie droite sont évalués.
- . Les attributs synthétisés du symbole en partie gauche sont évalués et communiqués au point de réduction.

La même remarque que précédemment, sur l'ordonnancement implicite, reste vraie.

Là encore, les implanteurs connaissent bien ces restrictions. Mais, bien plus que pour l'analyse descendante, des entorses sont faites à cette restriction pour hériter des informations, de gauche à droite dans une production via l'environnement global.

#### 11.4.4.4. Attributs sémantiques et traducteurs classiques.

Les attributs sémantiques apportent une clarification et une systématisation du flot d'information sémantique. Les restrictions que nous venons de voir sont très sévères, en particulier pour l'analyse ascendante, trop sévères pour être tolérées en pratique. Les dérogations, qui étaient rendues possibles par un environnement global, accessible et modifiable sans limite, sont impossibles et les implanteurs subissent, là, une lourde pénalité. Les preuves de cohérence, qui sont, bien sûr, applicables sur des définitions restreintes, ne sont pas des arguments suffisants pour les convaincre de l'intérêt de la méthode. On peut affirmer que l'évaluation sémantique en alternance avec une analyse ascendante n'est pas envisageable. Par contre, l'analyse descendante n'est que peu pénalisée. Ce résultat confirme une impression, que connaissent bien les implanteurs, qui est que l'analyse descendante est plus commode pour des inclusions sémantiques. Nous venons de montrer qu'elle est, non seulement plus commode, mais encore plus justifiée théoriquement.

### 11.5. Réduction du système d'équations.

Ce paragraphe présente deux aspects de la notion d'attribut qui permettent de réduire le système d'équations aux équations utiles ne contenant que des inconnues utiles.

Nous avons déjà signalé que nous ne construisions pas effectivement un tel système à cause de son encombrement mais il est plus commode de supposer son existence pour la clarté de l'exposé. Cette réduction du système (ou des relations qui permettraient de le construire) a des effets rétroactifs sur les phases de construction et d'analyse, et nous décrivons sommairement les modifications à apporter dans ces phases pour que la réduction soit possible.

On doit remarquer que cette réduction peut, avec quelques modifications, s'appliquer à un système d'équations plus général et, en particulier, à la résolution de programmes LAU.

#### 11.5.1. *Elimination des identités.*

Définition 1 : une équation du système est une identité si et seulement si cette équation est de la forme  $x_i = x_j$

Définition 2 : une définition sémantique est une identité et nous la notons

$a(X_{ijk}) \equiv b(X_{ijl})$  si et seulement si, dans le système de

définition SE(i,j) on a  $a(X_{ilk}) := b(X_{ijl})$

pour  $0 \leq k, 1 \leq n_{ij}$   $a \in A(X_{ijk})$

$b \in A(X_{ijl})$

Le nombre d'identités est très élevé dans une définition sémantique (58 % des définitions pour SIMULA 67). Ceci est une conséquence de la contrainte de localisation des définitions. Une information élaborée dans une règle doit être transmise dans l'arbre partout où elle est utile et cette transmission correspond à une cascade d'identités.

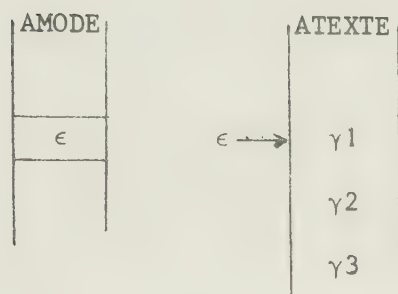
Il paraît donc souhaitable d'éliminer ces identités pour ne garder que les inconnues utiles.

Ce traitement nécessite une intervention du constructeur qui repère les identités et les signale pour les phases ultérieures.

##### 11.5.1.1. Traitement à la construction.

Pour une identité, il n'y a pas création d'un modèle d'évaluation sémantique. Pour continuer à avoir une représentation uniforme, un niveau de nomenclature supplémentaire est introduit : pour une règle, le tableau

AMODE donne une entrée dans un tableau ATEXTE comme suit :



Lors de la rencontre d'une identité, l'entrée correspondante de ATEXTE est vide.

#### 11.5.1.2. Traitement à l'analyse.

Nous ne voulons pas créer d'équations pour les identités, mais il nous faut un mécanisme pour les enregistrer, et répercuter dans les équations non identiques une numérotation convenable des inconnues. Aussi, nous allons modifier l'opération de décoration : tous les attributs d'un symbole n'ont plus le même indice mais sont indépendants les uns des autres ; ils ont leur indice propre. Reprenons l'exemple des paragraphes 11.3.1. et 11.3.2.

Nous avons par exemple,

|            |    |    |
|------------|----|----|
| opérande   | 12 | 13 |
| pm         | -  | -  |
| expression | 5  | 6  |
| syntaxe    | r  | g  |

chaque attribut de chaque symbole a son numéro. Lors de la réduction, un nouveau sommet de pile est créé et des numéros sont affectés aux attributs de ce nouveau sommet de la manière suivante :

|            |       |       |
|------------|-------|-------|
| expression | $n_1$ | $n_2$ |
|------------|-------|-------|

- $n_1$  et  $n_2$  sont des nouveaux numéros s'il n'y a aucune identité,
- $n_1$  et/ou  $n_2$  sont les numéros des attributs de pile s'il y a une identité.

Explicitons ce processus sur l'exemple complet du § 9.5.3. (les numéros entre parenthèses correspondent à la décoration de ce paragraphe).

## L'analyseur reconnaît successivement

|                |    |                | r | g  | équation                                                                               |
|----------------|----|----------------|---|----|----------------------------------------------------------------------------------------|
| id(1)          | en | opérande(2)    | 1 | 2  | $r_1 := \text{si } g_2 \text{ alors } 1 \text{ sinon } 0$                              |
| opérande(2)    | en | expression(3)  | 1 | 2  | -                                                                                      |
| + (4)          | en | pm(5)          | - | -  | -                                                                                      |
| id(6)          | en | opérande(7)    | 3 | 4  | $r_3 := \text{si } g_4 \text{ alors } 1 \text{ sinon } 0$                              |
| opérande(7)    | en | expression(8)  | 3 | 4  | -                                                                                      |
| - (9)          | en | pm(10)         | - | -  | -                                                                                      |
| id(11)         | en | opérande(12)   | 5 | 6  | $r_5 := \text{si } g_6 \text{ alors } 1 \text{ sinon } 0$                              |
| expression(8)  |    |                |   |    |                                                                                        |
| pm (10)        | en | expression(14) | 7 | 8  | $g_4 := \text{vrai}$                                                                   |
| opérande(12)   |    |                |   |    | $g_6 := \text{faux}$                                                                   |
|                |    |                |   |    | $r_7 := \text{si } r_3 = r_5 \text{ alors } r_3 + 1$<br>$\text{sinon } \max(r_3, r_5)$ |
| ( (13)         |    |                |   |    |                                                                                        |
| expression(14) | en | opérande(16)   | 7 | 8  | -                                                                                      |
| ) (15)         |    |                |   |    |                                                                                        |
| expression(3)  |    |                |   |    |                                                                                        |
| pm (5)         | en | expression(17) | 9 | 10 | $g_2 := \text{vrai}$                                                                   |
| opérande(16)   |    |                |   |    | $g_8 := \text{faux}$                                                                   |
|                |    |                |   |    | $r_9 := \text{si } r_1 = r_7 \text{ alors } r_1 + 1$<br>$\text{sinon } \max(r_1, r_7)$ |
| expression(17) | en | programme(18)  | 9 | -  | $g_{10} := \text{vrai}$                                                                |

Il y a donc synthèse des numéros d'attributs synthétisés et anticipation puis synthèse des numéros d'attributs hérités.

En fait, comme on veut accepter des identités entre inconnues correspondant à des attributs génériques différents, on ne doit pas conserver le nom générique mais le remplacer par un nom standard, par exemple  $x$  ce qui donne, pour l'exemple précédent :

```

x1 := si x2 alors 1 sinon 0
x3 := si x4 alors 1 sinon 0
x5 := si x6 alors 1 sinon 0
x4 := vrai
x6 := faux
x7 := si x3 = x4 alors x3 + 1 sinon max(x3, x4)
x2 := vrai
x8 := faux
x9 := si x1 = x7 alors x1 + 1 sinon max(x1, x7)
x10 := vrai

```

Ce traitement, extrêmement simple, permet de réduire considérablement le nombre d'inconnues du système.

Cette réduction dépend, bien sûr, de la définition sémantique et du programme traité, elle est de l'ordre de 60 % en moyenne.

On doit remarquer que les identités sont, pour la plupart, la conséquence de l'utilisation d'une syntaxe concrète et non abstraite car, dans l'arbre d'analyse, les noeuds inutiles ont essentiellement des attributs identiques.

Un exemple convaincant est la production

```
<ins> ::= ( <liste> )
```

dans TURINGOL dont tous les attributs sont identiques.

Cependant, dans toutes les productions, il y a des identités, ce qui prouve que l'utilisation d'une syntaxe abstraite n'éviterait pas cette élimination. D'autre part, elle imposerait une transformation arbre concret-arbre abstrait qui serait toujours plus coûteuse que le traitement très simple des identités au cours de l'analyse syntaxique.

#### 11.5.1.3. Limites du traitement à l'analyse.

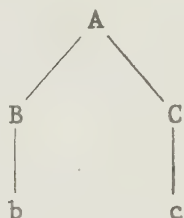
Le traitement décrit précédemment permet-il d'éliminer toutes les identités ? Malheureusement non. Il y a deux cas où il est insuffisant :

1er cas : la numérotation est effectuée au cours d'une analyse ascendante.

Considérons la définition suivante :

$$\begin{array}{ll}
 A \longrightarrow B \ C & a(C) \equiv a(B) \\
 B \longrightarrow b & a(B) = 0 \\
 C \longrightarrow c & b(C) = \dots a(C) \dots
 \end{array}$$

qui correspond à la ramification suivante :



L'analyse va s'accompagner d'une certaine numérotation, par exemple :

$$\begin{array}{llll}
 b & \text{en} & B & x_n = 0 \\
 c & \text{en} & C & x_m = \dots x_p \dots \\
 B & & & \\
 C & \text{en} & A & \text{on ne peut exprimer que } n = p
 \end{array}$$

Dans ce cas, on a anticipé de façon erronée dans la numérotation d'un attribut hérité et on ne peut défaire ce qui a été fait.

2ème cas :

$$\begin{array}{l}
 A \longrightarrow B \\
 c(A) \equiv a(A) \\
 d(B) \equiv b(B) \\
 a(A) \equiv b(B)
 \end{array}$$

Si les identités sont traitées dans cet ordre, on crée la numérotation

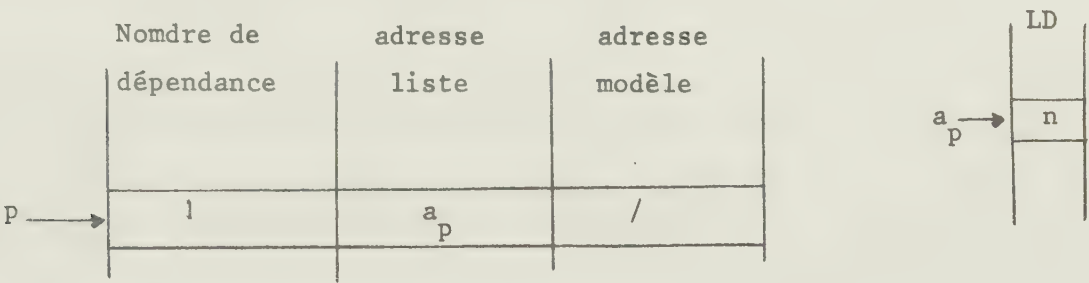
$$\begin{array}{ll}
 n & \text{pour } c(A) \text{ et } a(A) \\
 \text{puis } m & \text{pour } d(B) \text{ et } b(B) \\
 \text{et on ne peut exprimer que } & n = m
 \end{array}$$

Ce cas est plus simple à traiter que le premier et peut être résolu par ordonnancement local préalable, toujours réalisable si la définition est cohérente.

Le premier cas doit être traité à l'évaluation mais se présente très rarement (moins de 1 % des équations, en moyenne).

11.5.1.4. Traitement à l'évaluation.

Nous avons vu que le premier cas ne peut pas être traité complètement à la phase d'analyse. Une telle identité est traitée à l'évaluation et l'évaluateur en est averti par une information de la forme



L'absence d'adresse modèle signale qu'en fait, l'inconnue p est identique à l'inconnue figurant en a<sub>p</sub> dans LD.

11.5.2. *Elimination des impasses.*

11.5.2.1. Définitions, notations.

Définition 1 : on appelle IB, pour Inconnues Buts, l'ensemble défini par

$$IB = IS \cap \{a \mid a \in S(Z)\}$$

les inconnues-buts correspondant aux attributs synthétisés de l'axiome.

Notation : on note DE\* la fermeture transitive réflexive de la relation DE.

Définition 2 : on appelle IU, pour Inconnues Utiles l'ensemble défini par

$$IU = \{DE^*(a) \mid a \in IB\}$$

Définition 3 : on appelle II, pour Inconnues en Impasse, l'ensemble défini par

$$II = I - IU$$

où I est l'ensemble des Inconnues du système.

Les inconnues en impasse sont donc les inconnues qui ne participent pas à l'élaboration des inconnues finales du système, qui, rappelons le, correspondent aux attributs synthétisés de l'axiome de la grammaire.

Définition 4 : on appelle système réduit le système obtenu par élimination des équations de définition des attributs en impasse.

A ce système réduit, on associe un graphe réduit GR.

L'élimination des inconnues en impasse est justifiée par le fait que la "signification" d'un programme est exprimée à son axiome et, donc, toutes les inconnues qui ne servent pas à exprimer cette signification sont inutiles et peuvent être éliminées du système sans perte d'informations.

#### 11.5.2.2. Algorithme d'élimination des impasses.

On doit remarquer que si on applique un algorithme de parcours descendant en se limitant, au départ, aux inconnues-buts puis, de proche en proche, aux impliquants des inconnues visitées par la relation DE, les seuls inconnues visitées sont naturellement les inconnues utiles. Par exclusion, on peut donc déterminer et éliminer les inconnues, donc les équations inutiles.

L'algorithme de parcours descendant est décrit au paragraphe 11.6.2. et l'algorithme d'élimination des impasses s'en déduit immédiatement.

On doit noter que l'élimination explicite des impasses impose de parcourir deux fois le système et que cette opération est coûteuse en temps car proportionnelle au nombre de relations de dépendance, donc d'arcs du graphe de dépendance.

Cette constatation donne un intérêt tout particulier au parcours descendant qui ne visite naturellement que les inconnues utiles d'un système non réduit.

#### 11.5.2.3. Exemple.

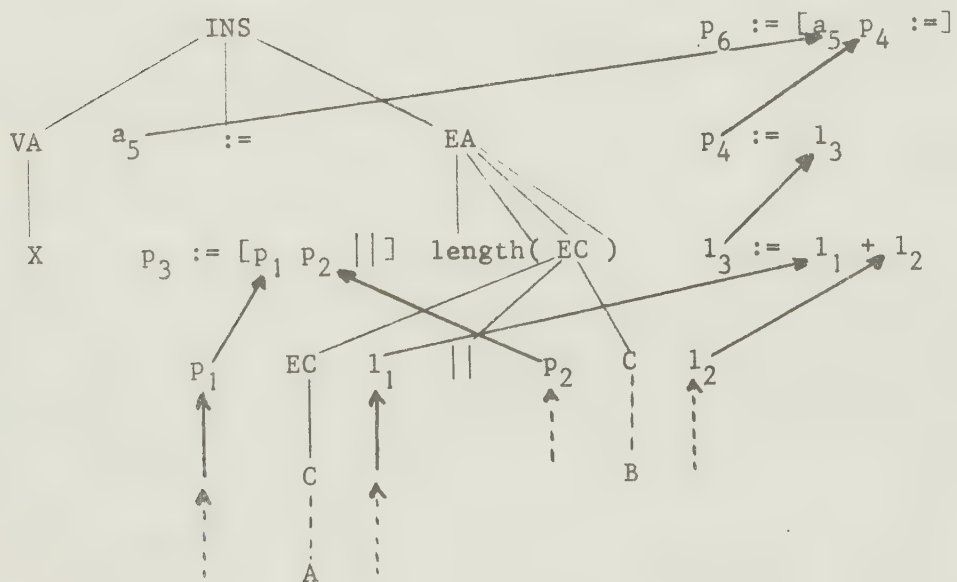
Prenons comme exemple d'application de l'élimination des impasses, la fonction PL/1 LENGTH ([39]) qui, appliquée à une expression de chaîne quelconque, retourne la longueur de cette expression de chaîne.

|                                        |                                  |
|----------------------------------------|----------------------------------|
| $INS \longrightarrow VA := EA$         | $p(INS) := [a(VA) \ p(EA) \ :=]$ |
| $INS \longrightarrow VC := EC$         | $p(INS) := [a(VC) \ p(EC) \ :=]$ |
| $EA \longrightarrow \text{length}(EC)$ | $p(EA) := l(EC)$                 |
| $EC \longrightarrow EC \    \ C$       | $p(EC) := [p(EC') \ p(C) \   ]$  |
|                                        | $l(EC) := l(EC') + l(C)$         |
| $EC \longrightarrow C$                 | $p(EC) := p(C)$                  |
|                                        | $l(EC) := l(C)$                  |

$||$  est l'opérateur de concaténation de chaînes,  $l$  est attribut de longueur d'une chaîne,  $p$  est un attribut de représentation polonaise post fixée.

Cet exemple est volontairement très simpliste ; un bon exercice pour le lecteur est de compléter cet exemple pour le rendre plus réaliste. Le lecteur se rendra compte de l'aspect très modulaire des définitions sémantiques.

L'arbre de dérivation de l'instruction  $X := \text{length}(A || B)$ , décoré par les attributs  $l$  et  $g$ , est le suivant :



#### 11.5.2.4. Impasses et définitions sémantiques.

La conséquence naturelle de l'élimination des impasses est l'élimination de tous les attributs liés à l'opération de concaténation. On peut comparer cette optimisation, obtenue comme une conséquence naturelle de l'aspect déclaratif de la méthode avec les solutions algorithmiques d'Elson et Rake [39]. La différence fondamentale vient, tout d'abord, d'une définition déclarative contre une solution algorithmique et, surtout, du fait que les différents concepts sémantiques attachés à une construction syntaxique, sont séparés. Que l'évaluation ne retienne que les concepts utiles à chaque occurrence particulière de chaque construction syntaxique est infiniment souhaitable.

Une règle de production, accompagnée de la définition des différents attributs qui expriment tous les concepts sémantiques, doit bien être considérée comme un modèle syntaxique et sémantique général. Mais ses occurrences constituent autant de cas spécifiques au contexte particulier où elles se présentent.

Ce problème, que connaissent bien les compilateurs, a fait dire que les compilateurs dirigés par la syntaxe sont naturellement pessimistes car tous les cas possibles de toutes les occurrences possibles doivent être envisagés.

Une évaluation sémantique sélective apporte une solution extrêmement séduisante à ce problème. Et il faut encore souligner qu'elle n'est le fruit d'aucun algorithme coûteux, c'est une conséquence normale et naturelle de la méthode.

Donnons quelques chiffres sur le nombre d'inconnues concernées par l'élimination des impasses :

les inconnues en impasses représentent entre 15 et 55 % des inconnues restant après élimination des identités.

L'étalement de ces pourcentages s'explique par le fait que le nombre d'impasses dépend étroitement de la définition du langage et du programme traité.

Une question philosophique est souvent posée : n'est-il pas préférable de s'attacher, dans la définition d'un langage, à exprimer le maximum de dépendance pour minimiser les impasses ?

La réponse est catégoriquement non, parce qu'une telle attitude va à l'encontre de la méthode des attributs. Rompre l'aspect déclaratif pour de présumés gains d'efficacité dans l'évaluation, porte un grave préjudice à

la clarté, la modularité des définitions. L'évaluation est et doit rester transparente à l'utilisateur du système et, en aucun cas, ne doit orienter la définition. Si nous ne suivons pas Fang dans une évaluation par processus "aléatoires" pour ce que cette solution apporte d'inefficacité, nous partageons entièrement son point de vue pour conserver totalement l'intérêt énorme de définitions déclaratives.

#### 11.6. Traitement à l'évaluation sémantique dans DELTA.

Ce paragraphe présente les algorithmes de parcours séquentiel du graphe de dépendance.

##### 11.6.1. *Définitions.*

On appelle tri la détermination d'un ordre d'évaluation des inconnues du système d'équations.

- . Un tri est donc une séquence des  $n$  inconnues du système, que nous notons

$$T = y_1 y_2 \dots y_n$$

- . Un tri est complet si chaque inconnue apparaît une fois et une seule dans la séquence.

- . Un tri est consistant si la séquence respecte les relations de dépendance entre inconnues, c'est-à-dire :

$$a \text{ DE } b \implies \exists i, j \quad 1 \leq i < j \leq n \mid y_i = b \text{ et } y_j = a$$

- . Nous ne nous intéressons, bien sûr, qu'aux tris consistants et complets pour les attributs utiles.

##### 11.6.2. *Tri descendant.*

Cet algorithme est décrit, tout d'abord, sous forme récursive avec les notations ensemblistes utilisées jusqu'à maintenant, puis une version fonctionnant à l'aide d'une pile est présentée, en utilisant la structure de données construite par l'analyseur.

###### 11.6.2.1. Tri descendant (première version)

début

inconnue  $X$  ;

procédure  $RP(X)$  ; inconnue  $X$  ;

début inconnue  $Z$  ;

pour  $Z \in DE [X]$  faire

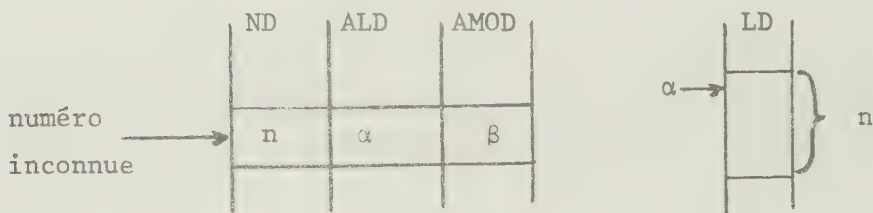
```

    si non EVALUE [Z] alors RP (Z) ;
commentaire : on traite les impliquants non évalués d'une inconnue ;
    EVAL (X) ;
    fin RP ;
procédure EVAL (X) ; inconnue X ;
    début
    EVALUE [X] := vrai ;
    ... commentaire : évaluation ou sortie du texte de l'équation ;
    fin EVAL ;
pour X ∈ S faire EVALUE [X] := faux ;
commentaire : toutes les inconnues du système sont non évaluées ;
pour X ∈ IB faire RP (X) :
commentaire : le tri commence sur les inconnues-buts ;
fin ;

```

#### 11.6.2.2. Tri descendant de DELTA.

Rappelons la représentation des équations construite par l'analyseur :



- . ND est le nombre de dépendance  $(|DE(x)|)$
- . ALD est l'adresse des impliquants dans LE ; on a donc
 
$$DE(x) = LD [\alpha : \alpha + n - 1]$$
- . AMOD est l'adresse du modèle d'évaluation.
- . Nous supposons, pour simplifier la présentation de l'algorithme, qu'il n'y a aucune identité dans le système.
- . Les inconnues-buts sont mémorisées dans le tableau B [1 : NB]
- . L'information EVALUE de l'algorithme précédent est donnée par ND :
 
$$\begin{aligned} ND \geq 0 & \quad \text{non EVALUE} \\ ND < 0 & \quad \text{EVALUE} \end{aligned}$$
- . La pile de traitement est PILE d'indice XP : on y mémorise le rang dans LD de l'impliquant traité.

Avec ces conventions, l'algorithme est le suivant (les déclarations de toutes les informations précédentes sont omises) :

début

procédure EVAL (X) ; entier X ;

début

ND [X] := -1 ;

...

fin EVAL ;

pour XB := 1 pas 1 jusqu'à NB faire

début

X := B [XB] ;

commentaire : but du tri ;

TD : Z := ALD [X] ;

commentaire : vers DE(X) ;

SUITE : si ND [X] > 0 alors

début

commentaire : X a des dépendants :

XP := XP + 1 ;

PILE [XP] := X, Z ;

commentaire : on conserve X et le pointeur vers DE(X) ;

X := LD [Z] ;

aller a TD ;

commentaire : traiter le dépendant courant ;

fin

sinon

si ND [X] = 0 alors

début

commentaire : on peut évaluer X ;

EVAL (X) ;

si XP = 0 alors aller a FIN

sinon début

X, Z := PILE [XP] ;

XP := XP - 1 ;

fin ;

fin ;

```

commentaire : on a traité le dépendant courant ;
ND [X] := ND [X] -1 ;
Z := Z +1 ;
commentaire : on passe au suivant ;
allera SUITE ;
FIN : fin ;
fin ;

```

Cet algorithme, qui nous a été suggéré par C. PAIR, est simple et efficace : il travaille en temps proportionnel au nombre de dépendance reliant deux inconnues utiles, car il ignore les inconnues en impasse. C'est donc l'algorithme de base de l'évaluation sémantique dans DELTA.

#### 11.6.3. *Tri ascendant.*

Comme pour le tri descendant, nous donnons deux versions de cet algorithme, l'une avec les notations ensemblistes, l'autre en simili-ALGOL sur les structures de données de DELTA.

##### 11.6.3.1. Tri ascendant (1ère version)

```

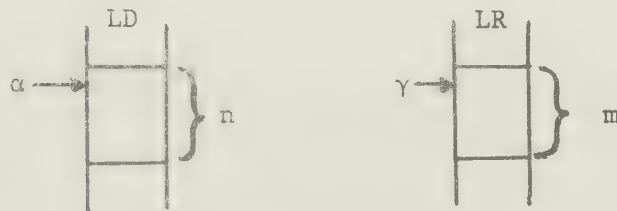
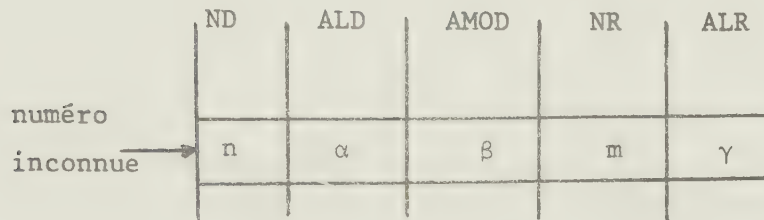
debut
  inconnue X, Y ;
  procedure EVAL(X) ; inconnue X ;
    debut
      EVALUE [X] := vrai ;
      ...
    fin EVAL ;
    pour X ∈ S faire EVALUE [X] := faux ;
    - commentaire : toutes les inconnues du système sont non évaluées.
      Le tri ascendant progresse en éliminant les inconnues sans impliquant.
      Mais ces inconnues sont nécessaires pour la procédure EVAL. On
      doit donc dupliquer avant tout traitement ;
      pour X ∈ S faire Q [X] := DE [X] ;
    tant que ∃ X ∈ S tel que Q [X] = ∅ et non EVALUE [X] faire
      debut
        EVAL(X) ;
        pour Y ∈ DE-1 [X] faire Q Y := Q [Y] - X ;
      fin ;
  fin ;

```

### 11.6.3.2. Tri ascendant de DELTA

Le tri ascendant utilise la relation  $DE^{-1}$ . Or nous n'avons vu jusqu'à maintenant que la relation DE. En fait, ce tri nécessite que les informations sur les impliquants se doublent d'informations sur les impliqués par une inconnue.

La représentation des inconnues est, dans ce cas :



- . ND , ALD, AMOD ont la même signification que précédemment
- . NR est le nombre de références ( $|DE^{-1}(x)|$ )
- . ALR est l'adresse des références dans LR ; on a donc  $DE^{-1}(x) = LR[\gamma:\gamma+m-1]$
- . l'algorithme utilise une LISTE dont nous ne décrivons pas ici la gestion.
- Nous utilisons les notations  $X \leftarrow LISTE$  pour extraire un élément de la liste et  $X \rightarrow LISTE$  pour y placer un élément.
- . NI est le nombre total d'inconnues.

debut

procédure EVAL (X) ; entier X ;

debut

...

NDEP [X] := -1 ;

...

fin EVAL ;

pour X := 1 pas 1 jusqu'à NI faire

si ND [X] = 0 alors X  $\rightarrow$  LISTE ;

```

commentaire : on place dans LISTE toutes les inconnues évaluables
               directement ;
pour X ← LISTE tant que LISTE non vide faire
  debut
  EVAL(X) ;
  pour I := 1 pas 1 jusqu'à NR [X] faire
    debut
    Y := LR [ALQ[X] + I - 1] ;
    ND [Y] := ND [Y] - 1 ;
    si ND [Y] = 0 alors Y → LISTE
    fin
  fin
fin ;

```

#### 11.6.4. Comparaison entre tri descendant et ascendant.

##### 11.6.4.1. Place mémoire

Le tri ascendant présente l'inconvénient de demander à l'analyse plus d'informations (NR, LR, ALR) qui doublent pratiquement la place nécessaire à la mémorisation des équations.

Donnons quelques estimations de cette place, en comptant, en moyenne, ND et NR égaux à 2.

|          |        |
|----------|--------|
| ND - ALD | 1 mot  |
| AMOD     | 1 mot  |
| LD       | 2 mots |
| NR - ALR | 1 mot  |
| - LR     | 2 mots |

Soit : 4 mots pour le tri descendant et 7 mots pour le tri ascendant, en moyenne, par équation.

On peut noter que ces chiffres sont pessimistes car, lorsqu'une inconnue n'a qu'un impliquant ou un impliqué, elle peut ne nécessiter que 2 mots, par la remarque suivante : on peut utiliser directement ALD ou ALR

| numéro     | ND | ALD | AMOD | NR | ALR |
|------------|----|-----|------|----|-----|
| inconnue → | 1  | nd  | β    | 1  | nr  |

où nd et nr sont directement les numéros des inconnues.

Le tri descendant a besoin d'une pile, le tri ascendant d'une liste : il y a, à peu près, égalité sur ce point car si la pile est, en moyenne, plus volumineuse que la liste, il n'y a aucun problème à la gérer, ce qui est loin d'être le cas pour la liste.

#### 11.6.4.2. Temps

Les deux tris sont en temps proportionnel au nombre de relations de dépendance du système d'équations. Mais le tri ascendant nécessite une consultation initiale de toutes les inconnues alors que l'information initiale du tri descendant est directement connue.

#### 11.6.4.3. Comparaison qualitative

Le tri descendant assure, par construction, l'élimination des impasses, élimination sur le système : nous l'appellerons élimination statique des impasses. Il offre un autre avantage qui apparaît lorsqu'on envisage une équation du type :

$x = \text{si condition alors } v1 \text{ sinon } v2$

Un tri descendant gauche droite va évaluer la condition et selon ce résultat va s'aiguiller sur  $v1$  ou  $v2$  alors qu'un tri ascendant calcule la condition  $v1$  et  $v2$ . (Ces remarques se retrouvent lorsque l'on parle d'évaluation descendante ou ascendante, l'analyse descendante permettant, elle aussi, une évaluation sélective).

Le gros intérêt de cette constatation est que le tri descendant peut effectuer une élimination dynamique des impasses,  $v1$  ou  $v2$  étant nécessairement une impasse dans le cas de cette conditionnelle, élimination qui vient s'ajouter à l'élimination statique dont nous avons déjà parlé. Cet aspect prend une importance particulière lors de la résolution des équations :

$$a = \sqrt{x}$$

$$b = \sqrt{-x}$$

$y = \text{si } x \geq 0 \text{ alors } a \text{ sinon } b$

qui est impossible si une élimination dynamique n'intervient pas.

Nous avons longuement insisté sur l'importance que nous attachions à cet aspect des attributs sémantiques de permettre l'adaptation naturelle des définitions sémantiques au contexte où elles apparaissent. Il est indiscutable que le tri descendant est essentiel à la mise en oeuvre de cette adaptation par l'élimination complète des impasses qu'elle permet naturellement.

Après avoir dit tant de bien du tri descendant, et nous avons justifié largement ce point de vue, nous défendrons tout aussi fort le tri ascendant au chapitre 12 car il met en évidence un aspect important pour des développements futurs de système à attributs : l'évaluation parallèle.

### 11.7. Espace mémoire nécessaire à l'évaluation sémantique

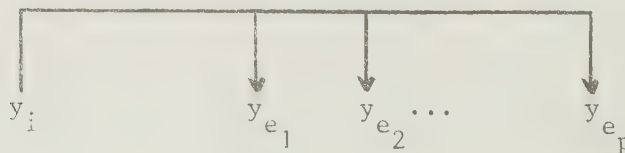
Dans ce paragraphe, nous nous intéressons à déterminer l'espace mémoire occupé par les valeurs des inconnues lors de la résolution du système. Nous avons considéré, jusqu'à maintenant, que chaque inconnue occupait une cellule mémoire qui n'était jamais partagée. Cependant la localisation des attributs fait qu'un attribut a une durée de vie limitée. On peut donc affirmer que lorsque la valeur d'une inconnue du système a été utilisée par toutes ses références, la cellule mémoire occupée par cette valeur devient récupérable et réutilisable pour une autre inconnue. Nous ne considérons ici que le cas où toutes les inconnues nécessitent la même taille de cellule-mémoire.

#### 11.7.1. *Définitions.*

Définition 1 : une allocation est une application des inconnues d'un système d'équations dans un ensemble d'adresses. L'allocation d'un tri détermine les adresses associées à chaque inconnue de la séquence du tri.

Définition 2 : une allocation est consistante si et seulement si  $1 \leq i < j < k \leq n$  et  $y_k \text{ DE } y_i \Rightarrow \text{allocation}(y_j) \neq \text{allocation}(y_i)$   
- on impose donc que l'allocation d'une inconnue n'est pas réutilisée tant que cette inconnue est référencée.

Notation : nous visualisons une allocation d'un tri par



qui exprime que  $\text{DE}^{-1}(y_i) = \{y_{e_j}, 1 \leq j \leq p\}$

Définition 3 : on appelle nombre d'inconnues vivantes en  $i$  et on note  $nv(i)$ , l'entier positif ou nul défini par

$nv(i) = \text{nombre d'inconnues } y_j \text{ avec } 1 \leq j < i$

$\mid \exists k \ i < k \leq n \text{ et } y_j \in DE^{-1}(y_k)$

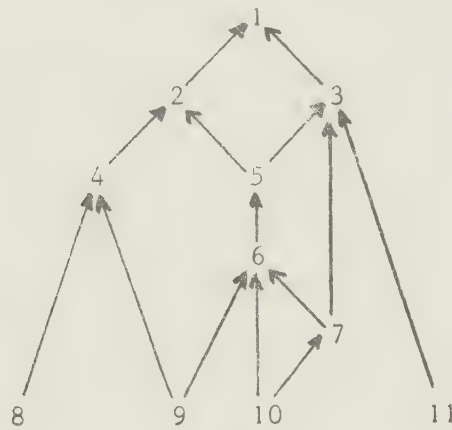
Ce nombre représente le nombre d'arcs de la notation précédente qui "passent au-dessus" de la position  $i$  du tri.

Définition 4 : on appelle taille d'un tri  $\max(nv(i), 1 \leq i \leq n) + 1$

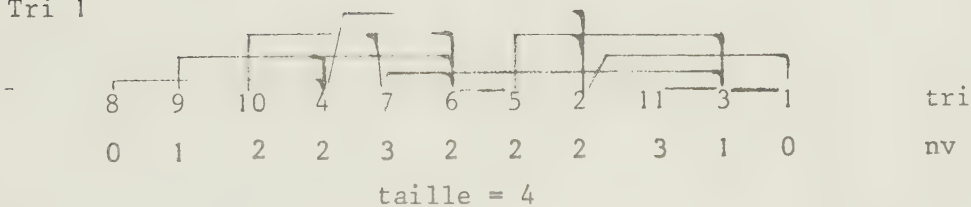
c'est le nombre minimum de cellule mémoire nécessaires à l'évaluation du système trié.

### 11.7.2. Exemple.

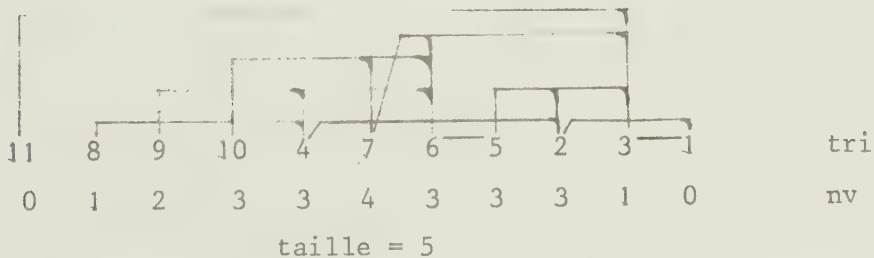
Donnons deux exemples pour éclairer les notions de  $nv$  et de taille.



Tri 1



Tri 2



Le deuxième tri est donc plus coûteux en place mémoire que le premier car sa taille est supérieure.

Décrivons les tris-allocations associés à ces deux exemples. Nous noterons  $X_i$  une cellule mémoire et  $x_j$  l'inconnue qui occupe  $X_i$ .

|    | TRI 1 | X1 | X2 | X3  | X4 |
|----|-------|----|----|-----|----|
| 1  | x8    | x8 |    |     |    |
| 2  | x9    | x8 | x9 |     |    |
| 3  | x10   | x8 | x9 | x10 |    |
| 4  | x4    | x4 | x9 | x10 |    |
| 5  | x7    | x4 | x9 | x10 | x7 |
| 6  | x6    | x4 | x6 |     | x7 |
| 7  | x5    | x4 | x5 |     | x7 |
| 8  | x2    | x2 | x5 |     | x7 |
| 9  | x11   | x2 | x5 | x11 | x7 |
| 10 | x3    | x2 | x3 |     |    |
| 11 | x1    | x1 |    |     |    |

|    | TRI 2 | X1  | X2 | X3 | X4  | X5 |
|----|-------|-----|----|----|-----|----|
| 1  | x11   | x11 |    |    |     |    |
| 2  | x8    | x11 | x8 |    |     |    |
| 3  | x9    | x11 | x8 | x9 |     |    |
| 4  | x10   | x11 | x8 | x9 | x10 |    |
| 5  | x4    | x11 | x4 | x9 | x10 |    |
| 6  | x7    | x11 | x4 | x9 | x10 | x7 |
| 7  | x6    | x11 | x4 | x6 |     | x7 |
| 8  | x7    | x11 | x4 | x5 |     | x7 |
| 9  | x2    | x11 | x2 | x5 |     | x7 |
| 10 |       | x3  | x2 |    |     |    |
| 11 |       | x1  |    |    |     |    |

### 11.7.3. *Solutions optimales.*

On peut souhaiter trouver un tri dont la taille soit minimale. Un problème voisin a été étudié par Sethi [40], celui de l'allocation optimale de registres dans un programme séquentiel. Si on considère que chaque inconnue occupe un registre, on voit l'analogie avec notre problème.

Un autre problème, de même nature, a été étudié par Ullman [41], celui de la répartition sur des processeurs, de tâches partiellement ordonnées dans le temps. Ces deux problèmes ont été montrés, dans les papiers référencés, être non déterministes polynomiaux, c'est-à-dire qu'on ne sait pas si on peut les résoudre en un temps inférieur à un temps exponentiel du nombre de données. Il n'a pas été montré que notre problème appartient à la même classe de complexité mais on a de bonnes raisons de le penser.

Nous avons déjà signalé que les objectifs de DELTA excluent d'appliquer des algorithmes combinatoires complexes. Nous devons nous limiter à des heuristiques simples, en acceptant que des solutions optimales ne soient pas atteintes. Les résultats obtenus sont, d'ailleurs, largement en faveur de tels choix.

#### 11.7.4. *Allocation des tris descendant et ascendant.*

##### 11.7.4.1. Introduction

Nous allons reprendre les algorithmes de tri du paragraphe 11.6 et y incorporer une allocation simple qui, à chaque appel de la procédure d'évaluation EVAL, va rechercher une cellule mémoire libre pour réaliser l'évaluation.

Nous ne cherchons donc pas à déterminer le tri en fonction de la taille, mais nous laissons le tri se dérouler, sans aucune intervention, et nous en déduisons sa taille. Nous donnons ensuite des heuristiques simples qui permettent d'orienter localement le tri pour réduire sa taille.

Le seul problème qui existe, avec ces hypothèses, est de déterminer si l'inconnue associée à une cellule mémoire est morte ou vivante. Or nous avons introduit, pour le tri ascendant, la notion de nombre de références, NR, qui nous donne précisément cette information ; dès que  $NR[x] = 0$ , l'inconnue  $x$  n'est plus référencée : elle est donc morte.

Dans les algorithmes, nous représentons l'ensemble des adresses par un tableau booléen L tel que

$L[i] \iff$  la cellule d'adresse  $i$  est libre.

A chaque inconnue est associée son adresse A qui donne une entrée dans L.

On peut remarquer, dans les algorithmes, l'instruction

$A[X] := I$  tel que  $L[I]$ .

Cette instruction recherche une cellule libre pour X. Cette recherche peut être envisagée sous deux formes :

- . on recherche depuis la première cellule
- . on utilise une nouvelle cellule à chaque fois.

La première solution impose une recherche dans l'ensemble des adresses mais les résultats expérimentaux montrent que cette recherche n'est jamais longue car il y a peu de cellules mémoires utilisées.

La deuxième solution impose l'utilisation d'un ramasse-miettes et, pour être efficace, elle nécessite une zone mémoire d'évaluation importante si on ne veut pas appeler trop souvent le ramasse-miettes. De plus, à chaque appel à ce programme de compactification, une remise à jour des références à l'ensemble des adresses est nécessaire.

La première solution semble donc meilleure.

#### 11.7.4.2. Tris avec allocation

Les algorithmes de tri restent les mêmes que précédemment. Seule la procédure EVAL est modifiée pour effectuer l'allocation.

Cette procédure modifiée est, pour les deux tris :

```

procedure EVAL(X) ; inconnue X ;
  debut inconnue Y ;
  EVALUE [X] := vrai ;
  pour Y ∈ DE(X) faire
    debut
      NR [Y] := NR [Y] - 1 ;
      si NR [Y] = 0 alors L [A[Y]] := vrai ;
    fin
  A [Y] := I tel que L [I] ;
  L [I] := faux ;
fin EVAL ;

```

Suivons l'exemple du paragraphe 11.7.2.

Tri descendant avec allocation :

|            |          |             |           |           |
|------------|----------|-------------|-----------|-----------|
| RP(1)      |          |             |           | mémoire   |
| RP(2)      |          |             |           | récupérée |
| RP(4)      |          |             |           |           |
| RP(8)      | EVAL(8)  | A [8] := 1  |           |           |
| RP(9)      | EVAL(9)  | A [9] := 2  |           |           |
| EVAL(4)    |          | A [4] := 1  | 1         |           |
| RP(5)      |          |             |           |           |
| RP(6)      |          |             |           |           |
| RP(10)     | EVAL(10) | A [10] := 3 |           |           |
| RP(7)      | EVAL(7)  | A [7] := 4  |           |           |
| EVAL(6)    |          | A [6] := 2  | 2 et 3    |           |
| EVAL(5)    |          | A [5] := 2  | 2         |           |
| EVAL(2)    |          | A [2] := 1  | 1         |           |
| RP(3)      |          |             |           |           |
| RP(11)     | EVAL(11) | A [11] := 3 |           |           |
| EVAL(3)    |          | A [3] := 2  | 2, 3 et 4 |           |
| EVAL(1)    |          | A [1] := 1  | 1 et 2    |           |
| taille = 4 |          |             |           |           |

Tri ascendant avec allocation :

|            |             |  |          |           |
|------------|-------------|--|----------|-----------|
|            |             |  |          | mémoire   |
|            |             |  |          | récupérée |
| EVAL(8)    | A [8] := 1  |  |          |           |
| EVAL(9)    | A [9] := 2  |  |          |           |
| EVAL(10)   | A [10] := 3 |  |          |           |
| EVAL(11)   | A [11] := 4 |  |          |           |
| EVAL(4)    | A [4] := 1  |  | 1        |           |
| EVAL(7)    | A [7] := 5  |  |          |           |
| EVAL(6)    | A [6] := 2  |  | 2 et 3   |           |
| EVAL(5)    | A [5] := 2  |  | 2        |           |
| EVAL(2)    | A [2] := 1  |  | 1        |           |
| EVAL(3)    | A [3] := 2  |  | 2,4 et 5 |           |
| EVAL(1)    | A [1] := 1  |  | 1 et 2   |           |
| taille = 5 |             |  |          |           |

#### 11.7.4.3. Tri allocation avec heuristiques simples.

Donnons quelques constatations qui vont nous donner quelques heuristiques simples.

En ce qui concerne l'allocation, le tri descendant est souvent meilleur que le tri ascendant. Ce résultat s'explique naturellement, car il permet de localiser l'avaluation dans le graphe, alors que le tri ascendant a tendance à la dispersion, ce qui crée des inconnues vivantes en plus grand nombre. Or, la définition, que nous avons donnée, de la taille d'un tri-évaluation montre tout l'intérêt de la localisation : en effet, le nombre de branches recouvrant les inconnues diminue avec la localisation.

Les heuristiques, que l'on peut introduire, doivent donc favoriser la localisation.

D'autre part, les règles d'allocation sur des arbres (et non sur des graphes, en général), recommandent de favoriser les noeuds fortement référencés [29]. Pour connaître le nombre global de références à un noeud, il faudrait un parcours complet du graphe. Nous avons vu que nous ne pouvions pas adopter une telle solution, trop coûteuse, aussi avons nous introduit une heuristique sur le nombre de références immédiates à un noeud. C'est le nombre de référence NR, pour un tri ascendant, et le nombre de dépendance ND, pour un tri descendant. Les modifications sur les algorithmes de tri sont, avec ces heuristiques :

. tri descendant : dans la procédure RP, compléter l'instruction

pour  $Z \in DE [X]$  faire  
par dans l'ordre décroissant de ND [Z]

. tri ascendant : dans le corps du programme, compléter l'instruction

tant que  $\exists X$  tel que  $Q [X] = \emptyset$  et non EVALUE [X] faire  
par dans l'ordre décroissant de NR(X)

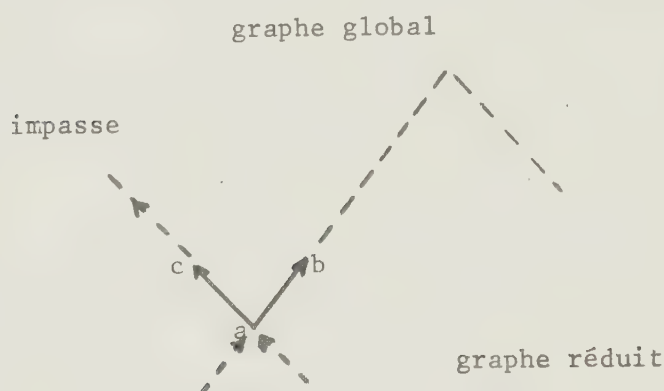
Il est difficile d'introduire simplement une contrainte sur la localisation de l'évaluation dans le tri ascendant. Tout au plus, peut-on ajouter à l'instruction précédente,

et pour les X dont le Q est devenu vide par la dernière évaluation.

Il s'avère que cette amélioration n'en est pas une. Elle n'apporte aucun gain sensible car elle est contradictoire avec la première.

#### 11.7.4.4. Tri allocation sur système non réduit.

Tous les algorithmes de parcours sont supposés travailler sur un système réduit. Nous avons signalé l'intérêt du tri descendant en profondeur qui évite naturellement les impasses d'un système global. Cet algorithme ne visite donc que les noeuds du graphe réduit GR. Mais il ne permet pas de récupérer complètement la place mémoire. Considérons le graphe suivant :



Le noeud, a , est un noeud d'articulation entre le graphe réduit et une impasse. Il a donc un nombre de références qui tient compte de la référence c .

Comme l'algorithme ne visite pas les impasses, c ne sera jamais évalué et a restera vivant jusqu'à la fin de l'évaluation. Une question mérite d'être posée : quel coût en espace mémoire représente ces noeuds d'articulation ? Est-il important de réduire le système avant l'évaluation ?

On doit noter que la place perdue est égale au nombre de noeuds d'articulation donc au nombre d'impasses. Sur les résultats expérimentaux du paragraphe suivant, la ligne NAR contient le nombre de noeuds d'articulation donc la perte de mémoire occasionnée par la non-réduction préalable au tri allocation. On peut voir que cette perte peut doubler ou tripler la place mémoire nécessaire à l'évaluation.

Une solution raisonnable, pour le tri descendant seulement, serait d'effectuer un tri sans allocation, la procédure EVAL se contentant de sortir le numéro de l'inconnue triée donc de déterminer les inconnues utiles ; puis de calculer le nombre de références NR de ces inconnues utiles, nombre qui est diminué en cas de références qui sont en impasse ; puis d'effectuer l'allocation sur la séquence triée.

Cette procédure est simple mais lourde car le recalcul des NR n'est pas immédiat.

Dans DELTA, une procédure directe d'élimination des impasses est implantée et activée à la demande de l'utilisateur du système. Cette procédure est indépendante du tri ultérieur. Mais il n'est pas certain que le gain obtenu soit intéressant par rapport au coût de mise en oeuvre de cette procédure.

#### 11.8. Résultats expérimentaux.

Les résultats qui suivent ont été obtenus sur un jeu de 4 définitions sémantiques. Ces résultats ne sont qu'expérimentaux, aussi ne permettent-ils pas des extrapolations hardies. Cependant, nous pensons qu'ils sont significatifs dans la mesure où ils proviennent d'exemples très diversifiés. Il faudra attendre une utilisation plus poussée du système DELTA pour avoir des résultats statistiques plus importants.

Ces exemples sont :

1 : Algorithme de SETHI-ULLMAN (§ 7.2) pour

$$a : A + (B - C)$$

$$b : A + (B - C / (D * E + F)) - (H - I)$$

2 : définition de TURINGOL (§ 7.2) pour le programme d'addition de 1 à un nombre binaire,

3 : définition d'un optimiseur local pour un sous ensemble de FORTRAN  
(2 programmes a et b),

4 : définition d'un optimiseur global pour le même sous ensemble de FORTRAN  
(2 programmes a et b)

Exemples.

|                                 | 1a       | 1b       | 2        | 3a        | 3b         | 4a        | 4b        |
|---------------------------------|----------|----------|----------|-----------|------------|-----------|-----------|
| . nombre d'inconnues            | 25       | 63       | 176      | 803       | 8604       | 137       | 492       |
| théorique (FOLDS)               |          |          |          |           |            |           |           |
| après élimination identités     | 10       | 30       | 51       | 274       | 2491       | 59        | 213       |
| après élimination impasses      | 8        | 23       | 28       | 128       | 1263       | 51        | 183       |
| noeuds d'articulation           | 0        | 0        | 0        | 27        | 198        | 2         | 6         |
| . taille                        |          |          |          |           |            |           |           |
| Tri descendant                  | 3<br>88% | 5<br>92% | 9<br>94% | 18<br>97% | 92<br>98%  | 23<br>83% | 78<br>84% |
| Tri descendant avec heuristique | 2<br>92% | 3<br>95% | 9<br>94% | 14<br>98% | 88<br>98%  | 21<br>85% | 91<br>81% |
| Tri ascendant                   | 3<br>88% | 8<br>87% | 9<br>94% | 30<br>96% | 320<br>96% | 19<br>86% | 63<br>87% |
| Tri ascendant avec heuristique  | 3<br>88% | 8<br>87% | 9<br>94% | 27<br>96% | 281<br>96% | 15<br>89% | 51<br>88% |

(Les pourcentages donnent les gains par rapport au nombre théorique)

Ces chiffres montrent que des algorithmes simples permettent d'améliorer considérablement les performances d'un système bâti autour de la notion d'attribut sémantique.

Les résultats précédents montrent aussi que les algorithmes de réduction de l'espace mémoire sont suffisants. Par rapport au système FOLDS, l'espace mémoire de l'évaluation est, dans DELTA, réduit de plus de 80 %.

Ce chiffre justifie nos algorithmes qui, s'ils ne donnent pas l'optimal, travaillent en temps linéaire du nombre d'arcs du graphe de dépendance. Une solution optimale n'apporterait qu'un gain supplémentaire faible pour une complexité et un temps de traitement sans commune mesure. Dans nos choix, il a fallu décider jusqu'où ne pas aller trop loin, pour que, globalement, les performances du système restent satisfaisantes.

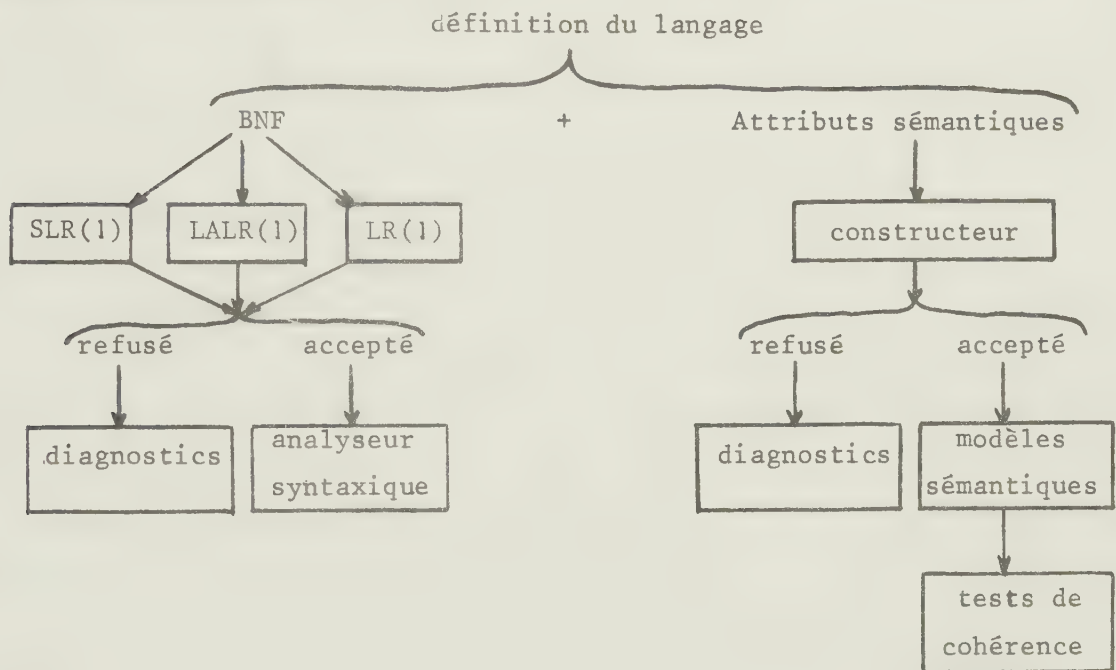
La comparaison entre l'exemple 3 et l'exemple 4 est pleine d'enseignements sur l'importance de la localisation des attributs. Ainsi, la définition d'un optimiseur local entraîne-t-elle une réduction de l'ordre de 97 %, alors que l'optimisation globale qui utilise naturellement des inconnues à longue durée de vie n'apporte-t-elle qu'un gain de l'ordre de 85 %.

### 11.9. Mise en oeuvre de DELTA

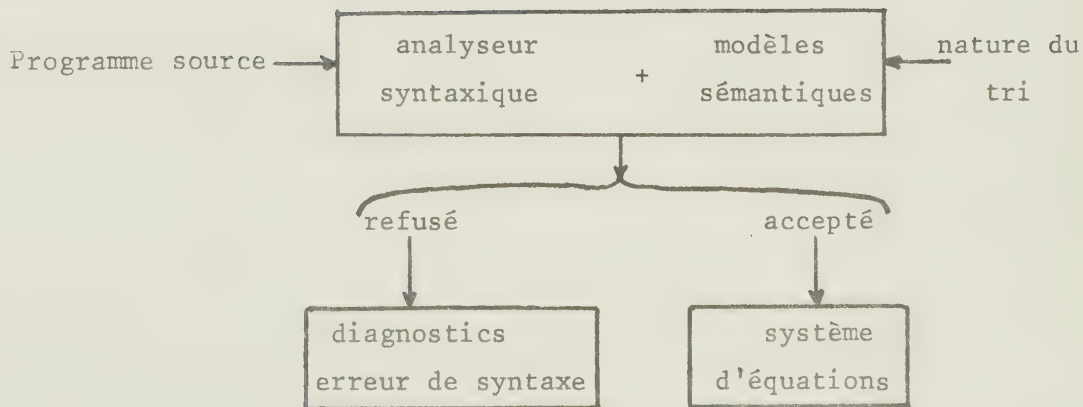
Nous allons décrire rapidement la mise en oeuvre du système DELTA pour les trois phases de construction, d'analyse et d'évaluation.

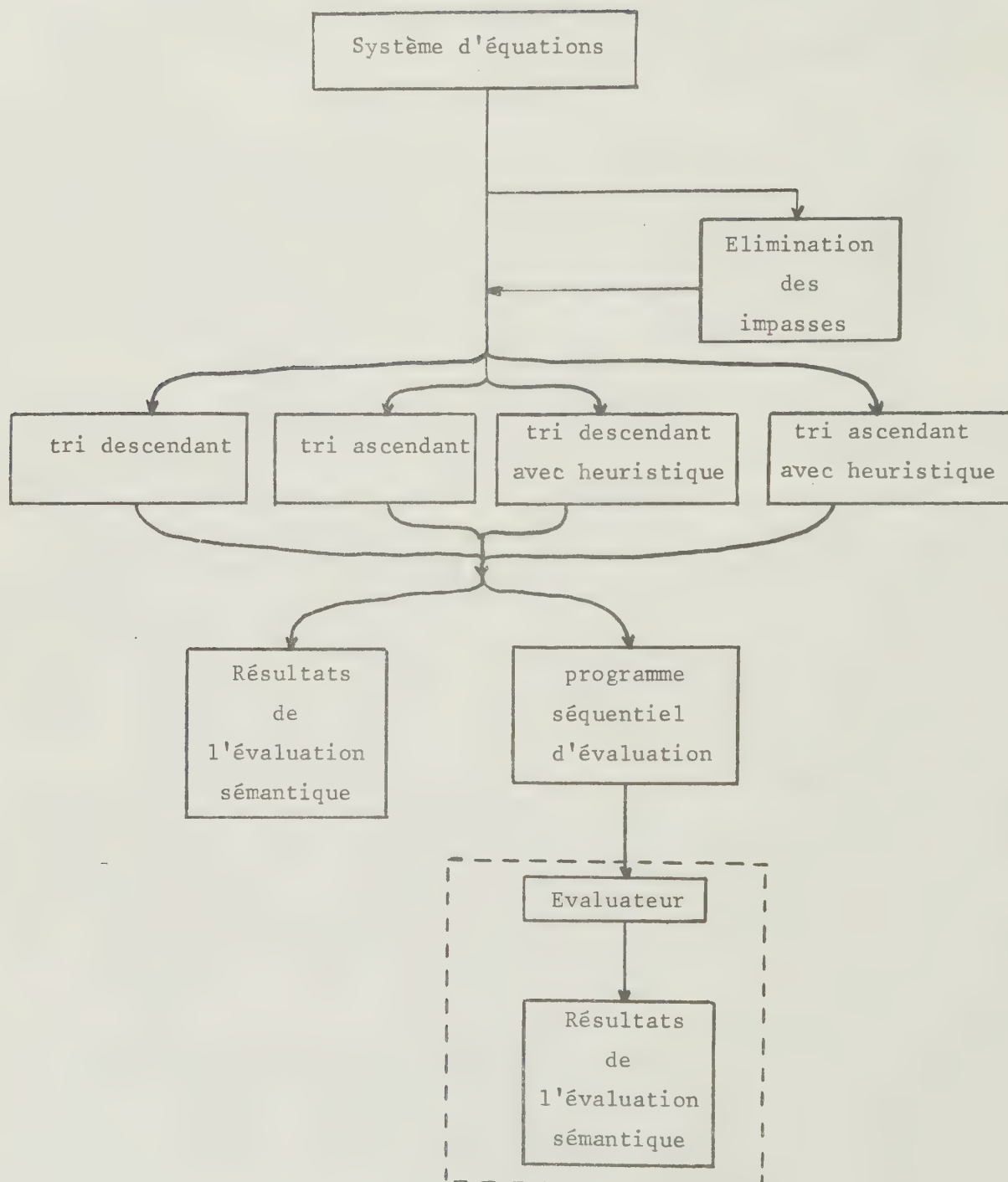
#### 11.9.1. Présentation fonctionnelle.

##### Phase construction :



##### Phase analyse :



Phase évaluation :

### 11.9.2. *Restriction actuelle.*

Les procédures de tri-allocation ne permettent, actuellement, que la production d'un programme séquentiel d'évaluation qui doit être exécuté en dehors du système. Cette restriction sera levée après la définition d'un langage de description des équations sémantiques et la réalisation d'un évaluateur de ce langage qui sera activé lors de l'appel à la procédure EVAL.

Alors, le tri-allocation sera remplacé par un tri-allocation-évaluation, l'allocation assurant la gestion des valeurs des inconnues résolues.

### 11.9.3. *Caractéristiques du système.*

Le découpage fonctionnel en trois phases correspond à un découpage réel du système en trois sous systèmes indépendants. Deux autres modules sont optionnels dans le système et peuvent être activés à la demande de l'utilisateur : les tests de cohérence et l'élimination des impasses.

La séparation des phases d'analyse et d'évaluation permet de ne pas construire effectivement le système d'équations sémantique à l'analyse, ce qui représente un gain de place appréciable.

Cette séparation offre également l'avantage de ne pas figer le système quant au type d'analyse syntaxique. Il est possible, sans aucune difficulté, de changer le constructeur syntaxique, donc l'analyseur syntaxique sans modification de la procédure de décoration.

De plus, la localisation des définitions sémantiques permet d'évacuer de la mémoire centrale les relations de dépendance, forme sous laquelle est mémorisé le graphe de dépendance, au fur et à mesure de leur élaboration, c'est-à-dire durant le traitement de la reconnaissance d'une règle.

Rappelons l'occupation mémoire moyenne des relations de dépendance liées à une équation : 4 mots auxquels il nous faut ajouter 1 mot pour le nombre de référence NR, soit 5 mots. On peut estimer qu'à une règle sont associées 3 équations.

Chaque occurrence de la ramification associée à une règle coûte donc, en moyenne 15 mots. Si on alloue une zone de mémorisation de 15000 mots, nous pouvons donc traiter des programmes correspondant à des ramifications de 1000 noeuds sans faire intervenir la procédure de sauvegarde sur support externe. Au delà, cette procédure est activée pour permettre de traiter des programmes de taille quelconque.

Ces estimations sont faites en vue d'un tri descendant. Pour un tri ascendant, il faut 8 mots, en moyenne, par équation soit environ 25 mots par règle : la procédure de vidage est activée tous les 600 noeuds.

Nous avons vu que le tri nécessite une pile ou une liste. La hauteur maximale de la pile est donnée par la longueur du plus long chemin entre une inconnue but et une inconnue d'entrée dans le graphe de dépendance. Elle peut donc être importante et c'est le seul ennui du tri descendant. Pour le tri ascendant, la longueur maximale de la liste est donnée par le nombre d'inconnues, dans le cas où toutes les équations sont évaluables immédiatement ! Dans la réalité, cette longueur ne dépasse jamais 100 mots.

Il nous est impossible de donner des estimations de temps sur le fonctionnement de DELTA parce qu'elles seraient faussées par le codage du système dans le langage ALGOL60 qui représente un certain coût par rapport à des langages de plus bas niveau, et surtout parce qu'aucun dispositif ne permet de le faire simplement sur notre calculateur.

Nous espérons que DELTA sera largement utilisé dans les universités et les centres de recherche français : il est disponible, non comme un produit industriel mais comme une maquette de recherche qui ne pourra que s'améliorer par un usage intensif.

## 12. COMPARAISON DES SYSTEMES A ATTRIBUTS, PERSPECTIVES D'AVENIR

### 12.1. Introduction.

Ce chapitre se propose de faire le point actuel des systèmes bâtis autour de la notion d'attribut sémantique. Puis nous reviendrons sur la notion d'assignation unique et parlerons des travaux faits dans ce domaine.

Enfin nous envisagerons les perspectives de développement de l'ensemble de ces systèmes sur une nouvelle structure de calculateurs, les multi-processeurs.

### 12.2. Systèmes à attributs.

De nombreux systèmes utilisent la notion d'attributs. Nous avons déjà cité FOLDS, de l'Université de Stanford, SOFT, de la Manufacture Belge de Lampes Electriques. Citons également les travaux des universités de Montréal et de Munich. Et nous avons vu que les grammaires affixes de KOSTER peuvent aussi entrer dans cette catégorie.

### 12.3. Comparaison de DELTA à FOLDS et SOFT.

Rappelons tout d'abord les objectifs de ces trois systèmes :

- . FOLDS est organisé pour la conception de langages de programmation,
- . SOFT vise à être un outil d'écriture de software,
- . DELTA essaie de concilier ces deux points de vue : son objectif est d'être un métacompilateur, où la description des langages et de leurs traducteurs est donnée le plus rigoureusement possible, dans l'état actuel des connaissances dans ce domaine.

#### 12.3.1. *Construction syntaxique.*

Sur ce plan, FOLDS et SOFT utilisent des variantes de l'algorithme d'Earley [49]. Dans FOLDS, on trouve des mécanismes très sophistiqués pour lever, sémantiquement, les ambiguïtés syntaxiques. Les objectifs de DELTA semblent justifier le choix de méthodes déterministes ; les comparaisons théoriques entre les méthodes déterministes ont décidé du choix des méthodes LR(k) et des raisons pratiques évidentes ont entraîné le choix de  $k = 1$ .

### 12.3.2. *Construction sémantique.*

DELTA est le seul système dans lequel les tests de cohérence sont implantés. Nous avons montré l'importance de ces tests d'un point de vue théorique et également pratique car ils entraînent un allègement considérable de l'évaluation sémantique par la suppression d'un grand nombre de tests dynamiques.

### 12.3.3. *Évaluation sémantique.*

Nous avons déjà décrit l'évaluation de FOLDS, par processus.

L'activation aléatoire de ces processus est un facteur de très grande inefficacité. SOFT effectue l'évaluation sémantique en alternance avec une analyse syntaxique descendante (cf §11.4.4.): nous avons montré que cette solution était la meilleure pour la traduction en un passage. DELTA exploite les relations de dépendance entre attributs : l'évaluation est dirigée par les attributs à évaluer, en temps proportionnel au nombre de relations de dépendance entre les attributs.

Signalons, d'autre part, que l'évaluation sémantique de FOLDS et SOFT ne peut se faire que sur l'arbre syntaxique complet ce qui correspond au système complet d'équations sémantiques : d'où une limitation importante sur la taille des programmes acceptés.

Enfin, seul DELTA contient des dispositifs d'élimination des identités et des impasses, ainsi qu'une procédure d'allocation de mémoire qui entraîne une réduction importante de l'espace mémoire nécessaire à l'évaluation.

### 12.4. Langages dirigés par les données.

Nous avons souligné la similitude entre évaluation sémantique et exécution de programmes en assignation unique. D'autres langages que les LAU ont été conçus dans la même optique, les langages dirigés par les données. L'exécution traditionnelle séquentielle des programmes n'a plus de sens, la seule règle pour diriger l'exécution est que les données nécessaires à une instruction sont connues.

Citons les travaux de Adams à Stanford [52], de Dennis au M.I.T. [53], de Roucairol à Paris 6, [54], ainsi que l'étude du CERT sur la définition d'un LAU [51].

### 12.5. Exécution séquentielle de programmes dirigés par les données.

On doit remarquer que toute la phase évaluation sémantique de DELTA pourrait être utilisée pour la séquentialisation de programmes dirigés par les données.

Il suffit de parcourir le programme en établissant les relations de dépendance sous forme de la structure de donnée présentée en 11.5. Tous les algorithmes de tri-allocation-réduction mémoire sont alors utilisables directement.

### 12.6. Exécution parallèle de programmes dirigés par les données.

L'intérêt essentiel de LAU et, plus généralement, de langages dirigés par les données est, bien sûr, les possibilités offertes d'exécution parallèle. En effet, où qu'elle soit située dans le programme, une instruction est exécutable dès que ses données sont connues. Et il peut y avoir un certain nombre d'instructions candidates à exécution. Si une structure de machine du type multiprocesseur est disponible, ces instructions peuvent s'exécuter en parallèle, chacune d'entre elles sur un processeur.

Les multi-processeurs ont longtemps été des machines extrêmement coûteuses mais la technologie a évolué et l'apparition de micro-processeurs de faible coût et de performances acceptables fait que cette structure de machines ne relève plus de l'utopie et peut devenir une réalité importante dans un avenir proche. On doit noter qu'une des raisons du retard dans le développement de ces matériels a été la difficulté d'en trouver des usages courants. Cette difficulté s'atténue avec les langages dirigés par les données.

Le problème de la répartition des tâches sur les différents processeurs est résolu par les algorithmes de tri ascendant qui se sont avérés être si médiocres pour la séquentialisation, mais qui retrouvent, dans ce cadre, une importance toute particulière.

Nous avons mesuré expérimentalement le degré de parallélisme, c'est-à-dire le nombre optimal de processeurs nécessaires pour une exécution parallèle idéale des exemples du paragraphe 11.8.

Le tableau suivant résume ces résultats en supposant que l'exécution se fait sur un système réduit d'équations et en supposant que la résolution de chaque équation prend une unité de temps identique pour toutes les équations.

|                                  | 1a | 1b | 2  | 3a | 3b  | 4a | 4b |
|----------------------------------|----|----|----|----|-----|----|----|
| Nombre optimal<br>de processeurs | 3  | 8  | 6  | 9  | 69  | 17 | 61 |
| nombre d'unité<br>de temps       | 4  | 8  | 12 | 47 | 412 | 18 | 62 |

Une étude de l'ordonnancement des tâches sur les processeurs lorsque leur nombre est inférieur au nombre optimal a été faite par Ullman [41]. Ce problème a été montré être non déterministe polynomial et une solution idéale est donc pratiquement exclue. Mais l'algorithme de tri ascendant peut être considéré comme une solution pratique raisonnable à ce problème, avec quelques aménagements possibles.

#### 12.7. Compilation parallèle.

Si nous acceptons l'hypothèse que les multi-microprocesseurs sont appelés à se développer, ils appelleront le développement de nouveaux langages dont les langages dirigés par les données font indiscutablement partie. Et la méthode des attributs sémantiques pourrait être, naturellement, le support privilégié de compilateurs parallèles car on ne voit guère, actuellement, d'autres méthodes de traduction permettant un tel type de compilation. Et les inconvénients qu'elle entraîne, taille du système d'équations et temps de résolution, seront, peut-être, largement compensés par ses qualités intrinsèques de mise en évidence du parallélisme des processus de traductions.

## CONCLUSION

*"Moi qui ai tant parlé, avec le désir insatiable de convaincre"*

Paul Valéry.

Au moment de conclure, il est certainement bon de rappeler les objectifs de cette thèse : rechercher la méthode de définition de langages de programmation la mieux adaptée au passage automatique aux traducteurs de ces langages ; analyser complètement cette méthode ; définir un système complet de traduction.

Toute la première partie consacrée à la description d'un certain nombre de méthodes ne prétend pas faire le tour complet de cette question. Nous avons essayé de dégager les caractéristiques et les objectifs essentiels de certaines d'entre elles et de les éclairer par la définition du même langage de programmation. Et en présentant les trois points de vue des théoriciens, des usagers et des implanteurs, nous espérons avoir fait une première synthèse dans un domaine qui en manque singulièrement. Il nous semble qu'ainsi nous avons placé le lecteur en meilleure position pour juger, en fonction de son appartenance à un ou plusieurs de ces trois groupes d'utilisateurs d'une définition, de la ou des méthodes les mieux adaptées à ses préoccupations.

Nous avons davantage insisté sur le point de vue des implanteurs car c'était notre objectif essentiel. Pouvons-nous affirmer avoir résolu le problème de la construction automatique de traducteurs ?

Ce serait de la témérité et de l'inconscience. Mais nous pensons avoir fait progresser ce domaine par le choix de la méthode des attributs sémantiques, car cette méthode est sous-jacente et implicite dans tous les traducteurs. Nous n'avons fait qu'explicitier des mécanismes d'évaluation et de contrôle d'informations sémantiques que connaissent bien les compilateurs, pour les manipuler quotidiennement. Nous n'avons pas rencontré que des avantages dans cette démarche, mais les preuves de cohérence sont un pas important vers des preuves de fiabilité de traducteurs et constituent l'apport marquant de la méthode. Les algorithmes que nous avons proposés sont corrects et réalistes dans la mesure où ils sont facilement implantables. L'inconvénient essentiel de la méthode est, bien sûr, la mémorisation du graphe de dépendance qui devient vite volumineux.

Mais nous avons montré que sa construction peut être faite partiellement, ce qui autorise à ne pas le conserver en mémoire dans sa totalité.

L'élimination des identités et des impasses offre le double avantage de réduire la taille du graphe, et de n'en conserver que ce qui est utile à l'évaluation sémantique d'un programme. C'est une solution élégante et naturelle au problème que connaissent bien les implanteurs : le pessimisme inévitable des description algorithmiques.

Nous proposons quatre algorithmes de parcours des graphes de dépendance, de complexité dépendant linéairement du nombre d'arcs de ces graphes. Ces algorithmes s'accompagnent d'une réduction importante de l'espace mémoire de l'évaluation sémantique.

Enfin, les perspectives d'avenir sur la compilation parallèle nous semblent fondamentales car elles éliminent un inconvénient de la méthode et la rendent tout à fait compétitive en temps, sinon peut-être plus performantes que les méthodes classiques. De plus, on voit mal comment ces méthodes classiques pourraient s'adapter à une évolution de la technologie vers des structures multi-processeurs, alors que notre méthode a tout à y gagner.

Avons-nous convaincu les implanteurs ? Nous l'espérons, en souhaitant que cette méthode place le traitement automatique de la sémantique dans une situation identique à celle qu'a connue la syntaxe lors de l'apparition des méthodes syntaxiques, il y a une dizaine d'années. Et ces méthodes, après une période de probation assez longue, sont, maintenant universellement adoptées.

Il nous faut maintenant aborder les développements futurs des systèmes à attributs, en général, et du système DELTA, en particulier.

La nécessité d'un langage de définition sémantique (LDS), pour la description des attributs, est évidente. FOLDS et SOFT contiennent de tels langages. Il n'y en a pas encore dans DELTA qui ne peut actuellement procéder que par création d'un programme d'évaluation sémantique traité en dehors du système.

La raison essentielle de cette absence est qu'il nous a semblé important de concevoir le système en dehors d'un langage particulier. Par exemple, SPINDLE, la langage de FOLDS, contient un certain nombre de dispositifs qui réagissent sur le processus d'évaluation. Cette approche nous semble criticable comme allant à l'encontre de la nature déclarative des attributs.

De ce point de vue, les langages utilisés dans DELTA pour la description sémantique ont été très variés : ALGOL60, FORTRAN, PL/1, LP70.

Nous en avons retiré la conviction qu'il fallait deux versions du système, l'une sans LDS où l'utilisateur a donc le libre choix de son LDS, le système ne lui offrant qu'une charpente syntaxique et sémantique ; l'autre où le système offre son LDS et des mécanismes complets d'évaluation. Parmi ces mécanismes, nous avons souligné l'importance du traitement des conditionnelles qui permet une élimination dynamique des impasses et est donc un facteur important d'amélioration des performances. Et on peut noter la grande fréquence d'apparition de ce concept dans les définitions du chapitre 8.

Quel sera le LDS de DELTA ? Les quelques points qui suivent seront à la base de sa définition :

- . aide apportée à l'utilisateur pour le choix de ses attributs, sous forme d'attributs standards,
- . variables structurées complexes pour la manipulation des structures de données du langage traité,
- . pas de variables globales,
- . axiomatisations et règles de déductions incorporées au LDS pour obliger l'utilisateur du système à une discipline de définition et pour compléter les algorithmes de cohérence.

Un autre axe de recherche à partir de DELTA est l'aspect structure de données du langage traité. Peu de méthodes ont été proposées pour ce problème ; la notion d'attribut peut certainement apporter beaucoup, pour la description statique du problème. D'autres mécanismes doivent être définis pour la description dynamique et nous pensons introduire un nouveau type d'attributs, dynamiques, c'est-à-dire ne respectant plus la contrainte de l'assignation unique. Quels seront ces attributs, comment interviendront-ils dans les algorithmes de cohérence ? Permettront-ils de compléter la gamme des méthodes consistantes entre elles, et complémentaires, pour que les trois catégories d'utilisateurs d'une définition soient également satisfaits ? Autant de questions qui méritent d'être posées et qui seront parmi nos prochains thèmes de recherche.

Dans le même souci de consistance et de complémentarité de méthodes, nous avons relevé quelques problèmes qui méritent une étude approfondie, en particulier les relations qui existent entre la méthode des attributs

sémantiques la méthode des doubles grammaires et les grammaires affixes.

Nous pensons aussi aux implanteurs non encore convaincus : une définition de la traduction du langage PASCAL [55] en P-code, langage d'une machine à pile, va commencer. Elle pourra être rapprochée de celle donnée par l'équipe de WIRTH.

L'utilisation de la notion d'attributs au problème de l'optimisation de code généré est déjà un thème de recherche du projet DELTA. Les résultats obtenus sont extrêmement encourageants ([56], [57]).

Citons une autre utilisation de cette notion, à l'interprétation d'APL. Elle a permis d'apporter une optimisation certaine à l'interprétation de ce langage ([58], [59]) et aussi d'offrir un support de comparaison entre les nombreuses méthodes de traitement d'APL [60].

Nous espérons que les universités et les centres de recherche utiliseront notre système. Il est à leur disposition et notre conviction est qu'il peut apporter beaucoup, comme outil pédagogique et comme système support de cours de traduction.

*"Les horizons aux horizons succèdent : on avance toujours, on n'arrive jamais"*

Victor Hugo.

## BIBLIOGRAPHIE

- [1] D.E. KNUTH  
Semantics of context free languages
  - a - Math-Systems Theory Vol 2 n° 2 p 127-145 (1968)
  - b - Vol 5 n° 1 p 95-96 (1971)
  
- [2] P. NAUR  
Revised report on the Algorithmic Language ALGOL 60  
CACM vol 6 n° 1, p 1-17 (1963)
  
- [3] J.W. BACKUS  
The syntax and semantics of the proposed international algebraic language  
Conference UNESCO Paris (1959)
  
- [4] M. GRIFFITHS  
Relationship between definition and implementation of a language  
Advanced course on software engineering  
Springer Verlag p 77-110 (1973)
  
- [5] C.H.A. KOSTER  
Affix grammars  
ALGOL 68 Implementation  
North Holland Pub. Co (1971)
  
- [6] D.E. KNUTH  
The remaining trouble spots in ALGOL 60  
CACM vol 10 n° 10 p 611-618 (1967)

- [7] J.V. GARWICK  
The definition of programming languages by their compilers  
Proc. IFIP Working Conf. p 266-294 (1964)
- [8] ECMA - ANSI PL/1-Basis 1/11 Mars 1974
- [9] N. WIRTH, H. WEBER  
EULER : a generalisation of ALGOL and its formal definition  
Part 1 : CACM vol 9 n° 1 p 13-25 (1966)  
Part 2 : CACM vol 9 n° 2 p 89-99 (1966)
- [10] P. LUCAS, K. WALK  
On the formal definition of PL/1  
Annual Review in Automatic Programming p 105-182 (1969)
- [11] P.E. LAUER  
Formal definition of ALGOL 60  
TR 25088 IBM Lab. Vienna (1968)
- [12] K. ALBER  
Syntactical description of PL/1 text and its translation into  
abstract normal form  
TR 25074 IBM Lab. Vienna (1967)
- [13] J.Mc CARTHY  
A basis for mathematical science of computation  
Formal programming languages,  
North Holland Pub. Co (1963)
- [14] J.Mc CARTHY et al.  
The LISP 1.5 programming manual  
MIT Press (1965)

[15] P. LANDIN

A Formal description of ALGOL 60  
Formal Languages description languages  
North Holland Pub. Co. (1966)

[16] C.A.R. HOARE, P.E. LAUER

Consistent and complementary formal theories of the semantics of  
programming languages.  
Acta Informatica vol 3, p 135-153 (1974)

[17] P. COUSOT

Définition interprétative et implantation des langages de programmation  
Thèse de Docteur-Ingénieur  
Université Scientifique et Médicale - Grenoble. (à paraître).

[18] C.A.R. HOARE

An axiomatic basis for computer programming  
CACM vol 12, n° 10 p 576-581 (1969)

[19] D. SCOTT

Outline of a mathematical theory of computation  
Princeton conference on Information Science and System  
p. 169-176 (1970)

[20] C. STRACHEY

Towards a formal semantics - Formal language description languages  
Proc. IFIP Working Conf. p 198-220 (1964)

[21] R.W. FLOYD

Assigning meaning to programs,  
Mathematical aspects of Computer Science (J. SCHWAZ edit.)  
Vol 19, p 19-32 (1967).

[22] J. ICHBIAH et al.

Manuel de référence du langage LIS - CII

- [23] Van WIJNGAARDEN (edit.)  
Revised report on the algorithmic language ALGOL 68 TR 74 - 3  
The University of Alberta, Edmonton (1974)
- [24] G. de CHASTELLIER, A. COLMERAUER  
W - Grammars  
Université de Montréal (document interne)
- [25] C.H.A. KOSTER  
Using the CDL compiler compiler  
Advanced course on compiler construction Munich (1974)
- [26] C. PAIR  
Cours d'analyse syntaxique  
Ecole d'Eté IRIA-CEA-EDF (1973)
- [27] D.E. KNUTH  
Top down syntax analysis  
International Summer School on Computer Programming - Copenhagen (1967)
- [28] D.E. KNUTH  
Examples of formal semantics  
Lecture notes in Mathematics n° 188, Springer Verlag (1971)
- [29] R. SETHI, J.D. ULLMAN  
The generation of optimal code for arithmetic expressions  
JACM vol 17 n° 4 p 715-728 (1970)
- [30] W.T. WILNER  
A declarative semantic definition  
PhD Thesis, Computer Science Department, Stanford University (1971)

- [31] I. FANG  
FOLDS, a declarative formal language definition system  
PhD Thesis, Computer Science Department, Stanford University (1972)  
et TR STAN CS-72-329 (1972)  
et Séminaire Structures et Programmation IRIA p 275-290 (1973)
- [32] M. BOUCKAERT, A. PIROTTE, M. SNELLING  
SOFT, a tool for writing software  
Report R212 MBLE Bruxelles (1973)  
et Séminaire Structures et Programmation IRIA p 291-315 (1973)
- [33] D. SCOTT, C. STRACHEY  
Towards a mathematical semantics for computer languages  
Technical monograph PRG-6 Oxford University (1971)
- [34] R. DACRE  
The mathematical and declarative meaning of computer languages  
Master of Science, University of Essex (1973)
- [35] D.E. KNUTH  
The art of computer programming vol 1 (1969)  
Addison Wesley Pub. Company
- [36] R. TARJAN  
Depth-First search and linear graph algorithms  
SIAM Journal of Computing vol 1 n° 2 p 146-160 (1972)
- [37] G.V. BOCHMANN  
Semantics evaluated from left to right  
Publication n° 135 Departement d'Informatique  
Université de Montréal (1973)
- [38] P.M. LEWIS, D.J. ROSENKRANTZ, R.E. STEARNS  
Attributed translations  
ACM Symposium on Theory of Computing p 160-171 (1973)

- [39] M. ELSON, S.T. RAKE  
Code generation technique for large language compilers  
IBM System Journal n° 3 p 166-187 (1970)
- [40] R. SETHI  
Complete register allocation problems  
ACM Symposium on Theory of Computing p 182-195 (1973)
- [41] J.D. ULLMAN  
Polynomial complete scheduling problems  
Princeton University
- [42] W.L. JOHNSON et al.  
Automatic generation of efficient lexical processors using  
finite state techniques  
CACM vol 11 n° 12, p 805-813 (1968)
- [43] F.L. DE REMER  
Lexical analysis  
Advanced course on compiler construction Munich (1974)
- [44] M. GRIFFITHS  
LL(1) grammars and analysers  
ibid
- [45] J.J. HORNING  
LR Grammars and analysers  
ibid
- [46] G. TERRINE  
Construction automatique d'analyseurs syntaxiques  
Thèse 3è cycle Université Scientifique et Médicale de Grenoble (1972)
- [47] F.L. DE REMER  
Practical translators for LR(k) languages  
PhD MIT (1969)

- [48] J.L. BOUCHENEZ, P. BOULLIER  
Projet DELTA : Constructeurs syntaxiques  
Rapport LABORIA (à paraître)
- [49] J. EARLEY  
An efficient context free parsing algorithm  
PhD Thesis, Computer Science Department,  
Carnegie Mellon University (1968)
- [50] D.D. CHAMBERLIN  
Parallel implementation of a single assignment language  
PhD Thesis, Computer Science Department,  
Stanford University (1971)
- [51] NICOLAS, SYRE  
Techniques et exploitation de l'assignation unique  
Doc 1/3044 CERT (1974)
- [52] D.A. ADAMS  
A computation model with data flow sequencing  
Tech. Rep. CS117  
Computer Science Department  
Stanford University (1968)
- [53] J.B. DENNIS  
First version of a data flow procedure language  
Séminaire Institut de Programmation (1974)
- [54] G. ROUCAIROL  
Transformation d'un programme séquentiel en programme  
parallèle parfait  
ibid

- [55] J.C. CLEAVELAND, R.C. UZGALIS  
What every programmer should know about grammars  
Modeling and Measurement Note n° 12  
Computer Science Department  
University of California, Los Angeles (1973)
- [56] M. AMIRCHAHY, M. MAZAUD, D. NEEL  
Optimisation de code généré par attributs  
Rapport LABORIA n° 59 (1974)
- [57] D. NEEL, M. AMIRCHAHY  
Semantic attributes and improvement of generated code  
ACM Conference San Diego (1974)
- [58] D. TUSERA  
Description d'une méthode d'interprétation du langage APL  
par les attributs sémantiques  
Thèse 3è cycle, Université de Paris 6 (1974)
- [59] D. TUSERA  
Example of transformation of a derivation tree for an  
expression by semantic attributes  
IFIP Congress (1974)
- [60] D. TUSERA, G. BATTAREL  
APL optimization : a unified approach  
APL Conference, Anaheim, Calif. (1974)
- [61] A. CHURCH  
The calculi of Lambda-Conversion  
Princeton (1941)

[62] D. SCOTT

Lattice Theory, Data Types and Semantics  
Formal Semantics of Programming Languages  
Prentice Hall, (1972) p 65-106

[63] S.K. ABDALL

A Simple Lambda-Calculus Model of Programming Languages  
AEC Research and Development Report C00-3077-28  
Courant Institute of Mathematical Sciences  
New York University (1973)

[64] C.A.R. HOARE, N. WIRTH

An axiomatic definition of the programming language PASCAL  
Berichte der fachgruppe computer  
Wissenschaften n° 6 (1972)

[65] M. CLINT, C.A.R. HOARE

Program proving : jumps and functions  
Acta Informatica 1 (1972) p 214-224

[66] N. CHOMSKY

On certain formal properties of grammars  
Information and Control vol 2 n° 2 p 137-167 (1957)



## ANNEXES



## ANNEXE 1

## TRADUCTION DE PROGRAMMES CONCRETS EN PROGRAMMES ABSTRAITS

Cette partie définit la traduction de programmes concrets TURINGOL en programmes abstraits satisfaisant la syntaxe abstraite du paragraphe 3.2.2. Cette traduction est décrite par attributs sémantiques, en utilisant un langage inspiré de VDL.

L'opérateur de construction élémentaire,  $\mu_0$ , est utilisé pour construire des objets VDL, simples ou composés :

Exemples .  $\mu_0 (<x:y>)$  construit un objet VDL simple, de sélecteur x et de valeur y

.  $\mu_0 (t , <x:y>)$  construit un objet VDL, composé de l'objet t et de l'objet simple de sélecteur x et de valeur y.

Les attributs utilisés sont :

|       |            |                                                         |
|-------|------------|---------------------------------------------------------|
| et    | hérité     | étiquettes déjà déclarées                               |
| étiq  | synthétisé | id                                                      |
| tape  | synthétisé | symboles de ruban déjà déclarés                         |
| n     | synthétisé | rang d'un symbole de ruban ou<br>rang d'une instruction |
| gtape | hérité     | ensemble des symboles de ruban                          |
| getiq | hérité     | ensemble des étiquettes                                 |
| ra    | synthétisé | représentation abstraite VDL                            |
| gd    | synthétisé | sens d'un déplacement                                   |

```

<programme> ::= <déclaration> ; <liste> .
    gtape(liste) := tape (déclaration)
    getiq(liste) := étiquetage(liste)
    ra(programme) :=  $\mu_0$  (<s-decl : ra(déclaration)>,
                          <s-corps : ra(liste)>)
    et(liste) :=  $\emptyset$ 

<déclaration> ::= TAPE ALPHABET IS id
    n(déclaration) := 1
    ra(déclaration) :=  $\mu_0$  (<elem(1) : text(id)>)
    tape(déclaration) := text(id)

<déclaration> ::= <déclaration> , id
    si text(id)  $\in$  tape(déclaration') alors
        erreur ("double déclaration")

    n(déclaration) := n(déclaration') + 1
    tape(déclaration) := tape(déclaration')  $\cup$  text(id)
    ra(déclaration) :=  $\mu_0$  (ra(déclaration') ,
                          <elem(n(déclaration)) : text(id)>)

<liste> ::= <instruction>
    et(instruction) := et(liste)
    etiq(liste) := etiq(instruction)
    gtape(instruction) := gtape(liste)
    getiq(instruction) := getiq(liste)
    ra(liste) := ra(instruction)
    n(liste) := 1

<liste> ::= <liste> ; <instruction>
    et(liste') := et(liste)
    et(instruction) := etiq(liste')
    etiq(liste) := etiq(instruction)
    gtape(liste') , gtape(instruction) := gtape(liste)
    getiq(liste') , getiq(instruction) := getiq(liste)
    ra(liste) :=  $\mu_0$  (ra(liste') , <elem(n(liste)) : ra(instruction)>)
    n(liste) := n(liste') + 1

```

```

<instruction> ::= <label> <ins>
    et(label) := et(instruction)
    et(ins) := etiq(label)
    etiq(instruction) := etiq(ins)
    gtape(ins) := gtape(instruction)
    getiq(ins) := getiq(instruction)
    ra(instruction) :=  $\mu_0$  (<s-label : ra(label)> ,
                                <s-ins : ra(ins)>)

<label> ::= id :
    si text(id)  $\in$  et (label) alors
        erreur("double définition d'étiquette")
    n(label) := 1
    ra(label) :=  $\mu_0$  (<elem(1) : text(id)>)
    etiq(label) := et(label)  $\cup$  text(id)

<label> ::= <label> id :
    et(label') := et(label)
    si text(id)  $\in$  etiq(label') alors
        erreur("double définition d'étiquette")
    n(label) := n(label') + 1
    etiq(label) := etiq(label')  $\cup$  text(id)
    ra(label) :=  $\mu_0$  (ra(label') , <elem(n(label)) : text(id)>)

<instruction> ::= <ins>
    et(ins) := et (instruction)
    etiq(instruction) := etiq(ins)
    getiq(ins) := getiq(instruction)
    gtape(ins) := gtape(instruction)
    ra(instruction) := ra(ins)

<ins> ::= PRINT id
    si text(id)  $\notin$  gtape(ins) alors
        erreur("symbole indéfini")
    etiq(ins) := et(ins)
    ra(ins) :=  $\mu_0$  (<s-type : PR> , <s-symbol : text(id)>)

```

```

<ins> ::= IF THE TAPE SYMBOL IS id THEN <instruction>
      si text(id) ≠ gtape(ins) alors
          erreur("symbole indéfini")
      et(instruction) := et(ins)
      etiq(ins) := etiq(instruction)
      getiq(instruction) := getiq(ins)
      gtape(instruction) := gtape(ins)
      ra(ins) := μ₀ (<s-type : COND> , <s-symbol : text(id)> ,
                    <s-ins : ra(instruction)>)

<ins> ::=
      etiq(ins) := et(ins)
      ra(ins) := μ₀ (<s-type : VIDE>)

<ins> ::= MOVE <sens> ONE SQUARE
      etiq(ins) := et(ins)
      ra(ins) := μ₀ (<s-type : MV> , <s-sens : dg(sens)>)

<sens> ::= LEFT
      gd(sens) := 1

<sens> ::= RIGHT
      gd(sens) := - 1

<ins> ::= GOTO id
      si text(id) ≠ getiq(ins) alors
          erreur("étiquette indéfinie")
      etiq(ins) := et(ins)
      ra(ins) := μ₀ (<s-type : GO> , <s-lab : text(id)>)

<ins> ::= (<liste>)
      et(liste) := et(ins)
      etiq(ins) := etiq(liste)
      getiq(liste) := getiq(ins)
      gtape(liste) := gtape(ins)
      ra(ins) := μ₀ (<s-type = COMP> , <s-corps : ra(liste)>)

```

Cet exemple illustre clairement la modularité des définitions par attributs, si on le compare à la définition du paragraphe 7.2 : pratiquement, la seule différence concerne l'attribut ra qui remplace l'attribut t1 et les attributs p et s qui ont disparu.

## ANNEXE 2

## INTERPRETEUR ABSTRAIT DE TURINGOL COMPLET

La prise en compte de l'instruction GOTO oblige à modifier notablement la définition du paragraphe 3.4 En effet, l'interprétation d'un branchement impose d'établir l'état du point de branchement. Or, TURINGOL autorise des branchements à n'importe quel point du programme, en particulier à l'intérieur d'une liste d'instruction. Il faut donc être capable, en recherchant l'étiquette de branchement, de construire la pile des compteurs d'instructions et d'établir la mémoire de contrôle pour que l'exécution se poursuive normalement.

La solution que nous proposons est de créer deux modes de parcours de l'arbre abstrait, le mode d'exécution vraie où le composant s-exec de l'état courant, est vrai, et le mode de pseudo exécution qui permet d'établir la mémoire de contrôle et la pile des compteurs d'instructions lors de la recherche d'une étiquette. L'interpréteur complet, avec ces conventions, est :

```
is-état = (<s-prog : is-programme> , <s-lib : is-lib> ,
          <s-c : is-control> , <s-tape : is-tape> ,
          <s-ci : is-pile> , <s-exec : is-bool> ,
          <s-lab : is-lab>)
```

Le composant s-lab donne l'étiquette à rechercher en mode de pseudo-exécution.

```

 $\xi_0 = \mu_0 ( \langle s\text{-prog} : \text{arbre abstrait} \rangle , \langle s\text{-lib} : \text{librairie} \rangle ,$ 
 $\langle s\text{-c} : \text{int-prog (arbre abstrait)} \rangle ,$ 
 $\langle s\text{-tape} : (\langle s\text{-case} : \text{ruban} \rangle , \langle s\text{-pos} : -1 \rangle) \rangle ,$ 
 $\langle s\text{-ci} : (\langle s\text{-s} : 1 \rangle , \langle s\text{-r} : \Omega \rangle) \rangle ,$ 
 $\langle s\text{-exec} : \text{vrai} \rangle , \langle s\text{-label} : \Omega \rangle )$ 

```

Librairie

```
int-prog(t)=int-linst(s-corps(t))
```

```
int-linst(t) =
```

```
    s-s(s-ci( $\xi$ ))  $\leq$  length(t)  $\rightarrow$  int-linst(t)
```

```
                                next
```

```
                                int-inst(elem(s-s(s-ci( $\xi$ ))))(t)
```

```
    vrai  $\rightarrow \Omega$ 
```

```
next =
```

```
    s-s(s-ci( $\xi$ )) : s-s(s-ci( $\xi$ )) + 1
```

```
int-inst(t) =
```

```
    is-inst-e(t) & s-exec( $\xi$ )  $\rightarrow$  tester-lab(s-lab( $\xi$ )) , t)
```

```
    is-inst-e(t)  $\rightarrow$  int-i(s-inst(t))
```

```
    vrai  $\rightarrow$  int-i(t)
```

```
int-i(t) =
```

```
    is-print-i(t)  $\rightarrow$  int-print(s-symbol(t))
```

```
    is-move-i(t)  $\rightarrow$  int-move(s)
```

```
                                s-sens(s-sens(t))
```

```
    is-vide-i(t)  $\rightarrow \Omega$ 
```

```
    is-cond-i(t)  $\rightarrow$  int-cond(s-symbol(t) , s-ins(t))
```

```
    is-comp-i(t)  $\rightarrow$  int-comp(s-corps(t))
```

```
    is-goto-i(t)  $\rightarrow$  int-goto(s-lab(t))
```

```
sens(t) =
```

```
    is-right(t)  $\rightarrow$  PASS:-1
```

```
    is-left(t)  $\rightarrow$  PASS:+1
```

```
int-print(s) =
```

```
    s-exec( $\xi$ )  $\rightarrow$  elem(s-pos(s-tape( $\xi$ )))(s-case(s-tape( $\xi$ ))) : s
```

```
    vrai  $\rightarrow \Omega$ 
```

```
int-move(i) =
```

```
    s-exec( $\xi$ )  $\rightarrow$  s-pas(s-tape( $\xi$ )) : s-pos(s-tape( $\xi$ )) + i
```

```
    vrai  $\rightarrow \Omega$ 
```

```

int-cond(s,t) =
    s-exec( $\xi$ ) &elem(s-pos(s-tape( $\xi$ )))(s-case(s-tape( $\xi$ ))) = s  $\rightarrow$  int-inst(t)
    s-exec( $\xi$ )  $\rightarrow \Omega$ 
    vrai  $\rightarrow$  int-inst(t)

int-comp(t) =
    vrai  $\rightarrow$  pop
        int-linst(t)
        push

push =
    s-ci( $\xi$ ) :  $\mu_0$ ( $\langle$ s-s=l $\rangle$  ,  $\langle$ s-r=s-ci( $\xi$ ) $\rangle$ )

pop =
    s-ci( $\xi$ ) : s-r(s-ci( $\xi$ ))

int-goto-i(lab) =
    s-exec( $\xi$ )  $\rightarrow$ 
     $\xi = \mu_0$ ( $\langle$ s-progr : arbre abstrait $\rangle$  ,  $\langle$ s-lib : librairie $\rangle$  ,
         $\langle$ s-c : int-prog (arbre abstrait) $\rangle$  ,
         $\langle$ s-tape : ( $\langle$ s-case : ruban $\rangle$  ,  $\langle$ s-pos : s-pos(s-tape( $\xi$ )) $\rangle$ ) $\rangle$  ,
         $\langle$ s-ci : ( $\langle$ s-s : l $\rangle$  ,  $\langle$ s-r :  $\Omega$  $\rangle$ ) $\rangle$  ,
         $\langle$ s-exec : faux $\rangle$  ,  $\langle$ s-label : lab $\rangle$  )
    vrai  $\rightarrow \Omega$ 

tester-lab(lab,t) =
    elem(i)(s-label(t)) |  $1 \leq i \leq \text{length}(s\text{-label}(t)) = \text{lab}$ 
         $\rightarrow$  int-i(s-inst(t))
        trouve

    vrai  $\rightarrow$  int-i(s-inst(t))

trouve =
    PASS :  $\Omega$ 
    s-exec( $\xi$ ) : vrai
    s-lab( $\xi$ ) :  $\Omega$ 

```

Autorisation de Soutenance accordée :

Toulouse le 12 Novembre 1974

Le Président

de l'Université Paul-Sabatier

L. LARENG.



